Стивен Прата

Рассматривается новый стандарт C++11

Язык программирования C++

6-е издание

ЛЕКЦИИ И УПРАЖНЕНИЯ

C++ Primer Plus

Sixth Edition

Stephen Prata

Addison
Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

Язык программирования C++

ЛЕКЦИИ И УПРАЖНЕНИЯ

6-е издание

Стивен Прата

Москва • Санкт-Петербург • Киев
2012

ББК 32.973.26-018.2.75
П70
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского *Ю.И. Корниенко, А А. Моргунова*

Под редакцией *Ю.Н. Артеменко*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Прата, Стивен.

П70 Язык программирования С++. Лекции и упражнения, 6-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2012. - 1248 с. : ил. - Парал. тит. англ.

ISBN 978-5-8459-1778-2 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Стивен Прата

Язык программирования С++. Лекции и упражнения 6-е издание

Верстка *Т.Н. Артеменко*

Художественный редактор *В.Г. Павлютин*

Подписано в печать 02.02.2012. Формат 70×100/16
Гарнитура Times. Печать офсетная
Усл. печ. л. 100,62. Уч.-изд. л. 72,3
Тираж 1000 экз. Заказ № 3024

Первая Академическая типография «Наука»
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1778-2 (рус.)
ISBN 978-0-321-77640-2 (англ.)

Оглавление

Содержание

Моим родителям, с любовью.

Благодарности

Благодарности к шестому изданию

Я хочу выразить благодарность Марку Тейберу (Mark Taber) и Саманте Синкхорн (Samantha Sinkhorn) из издательства Pearson за руководство и управление этим проектом, а также Девиду Хорвату (David Horvath) за техническое редактирование.

Благодарности к пятому изданию

Я хочу выразить благодарность Лоретте Йатс (Loretta Yates) и Сонглин Кию (Songlin Qiu) из издательства Sams Publishing за руководство и управление этим проектом. Я признателен своему коллеге Фреду Шмитту (Fred Schmitt), который дал мне несколько полезных советов. Спасибо Рону Личти (Ron Liechty) из компании Metrowerks за любезно предложенную помощь.

Благодарности к четвертому изданию

Организация и поддержка этого проекта была бы невозможной, если бы не участие редакторов из издательств Pearson и Sams. Я хочу выразить благодарность Линде Шарп (Linda Sharp), Карен Вачс (Karen Wachs) и Лори Мак-Гуайр (Laurie McGuire). Спасибо Майклу Мэддоксу (Michael Maddox), Биллу Крауну (Bill Craun), Крису Маундеру (Chris Maunder) и Филиппу Бруно (Phillipe Bruno) за техническое редактирование. Я еще раз благодарю Майкла Мэддокса и Билла Крауна (Bill Craun) за то, что они предоставили материалы для Real World Notes. В завершение я хочу выразить благодарность Рону Личти (Ron Liechty) из компании Metrowerks и Грегу Камю (Greg Comeau) из Comeau Computing за их помощь в работе с компиляторами C++.

Благодарности к третьему изданию

Я хочу поблагодарить редакторов из издательств Macmillian и The Waite Group за их участие в работе над этой книгой: Трейси Данкелбергер (Tracy Dunkelberger), Сьюзен Уолтон (Susan Walton) и Андреа Розенберг (Andrea Rosenberg). Спасибо Рассу Джейкобсу (Russ Jacobs) за техническое редактирование и пересмотр содержания книги. Также выражаю благодарность Дейву Марку (Dave Mark), Алексу Харперу (Alex Harper) и в особенности Рону Личти (Ron Liechty) за помощь и сотрудничество в подготовке этой книги.

Благодарности ко второму изданию

Я хочу поблагодарить Митчелла Уэйта (Mitchell Waite) и Скотта Каламара (Scott Calamar) за помощь в подготовке второго издания, а также Джоэла Фугацотто (Joel Fugazzotto) и Джоан Миллер (Joanne Miller) за руководство данным проектом вплоть до полного его завершения. Спасибо Майклу Маркотти (Michael Marcotty) из компании Metrowerks за то, что он не оставил без внимания ни один мой вопрос по третьей версии компилятора CodeWarrior. Я хочу выразить благодарность инструкторам, которые потратили свое время на организацию обратной связи по первому изданию: Джеффу Бакуолтеру (Jeff Buckwalter), Эрлу Бриннеру (Earl Brynner), Майку Холланду (Mike Holland), Энди Яо (Andy Yao), Ларри Сандерсу (Larry Sanders), Шахину Момтази (Shahin Momtazi) и Дону Стивенсу (Don Stephens).

Напоследок я хочу поблагодарить Хейди Брамбо (Heidi Brumbaugh) за помощь в редактировании нового и исправленного материала книги.

Благодарности к первому изданию

В работе над этой книгой принимали участие многие люди. В частности, я хочу поблагодарить Митча Уэйта (Mitch Waite) за его работу по составлению проекта, редактирование, а также за рецензию рукописи. Я признателен Гарри Хендерсону (Harry Henderson) за рецензирование нескольких последних глав и тестирование программ с использованием компилятора Zortech C++. Я благодарен Дэвиду Геррольду (David Gerrold) за рецензию всей рукописи, а также за его настойчивое требование учитывать интересы читателей, имеющих незначительный опыт в программировании.

Я благодарен Хэнку Шиффману (Hank Shiffman) за тестирование программ с использованием Sun C++ и Кенту Уильямсу (Kent Williams) за тестирование программ с помощью AT&T cfront и G++. Спасибо Нэну Борресону (Nan Borreson) из компании Borland International за то, что он любезно и энергично помог разобраться с Turbo C++ и Borland C++. Спасибо вам, Рут Майерс (Ruth Myers) и Кристин Буш (Christine Bush), за обработку непрекращающегося потока корреспонденции, связанной с этим проектом. В завершение я хочу поблагодарить Скотта Каламара (Scott Calamar) за то, что работа над этой книгой велась надлежащим образом.

Об авторе

Стивен Прата преподает астрономию, физику и вычислительную технику в морском колледже города Кентфилд, шт. Калифорния. Диплом бакалавра он получил в Калифорнийском технологическом институте, а степень доктора философии — в Калифорнийском университете в Беркли. Прата — автор и соавтор более десятка книг, включая *C Primer Plus* (*Язык программирования C. Лекции и упражнения, 5-е изд.*, ИД "Вильямс", 2006 г.) и *Unix Primer Plus.*

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш веб-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail: `info@williamspublishing.com`

WWW: `http://www.williamspublishing.com`

Информация для писем из:

России: 127055, г. Москва, ул. Лесная, д. 43, стр. 1

Украины: 03150, Киев, а/я 152

Введение

Процесс изучения языка программирования C++ чем-то напоминает приключение первооткрывателя, в частности потому, что этот язык охватывает несколько парадигм программирования, включая объектно-ориентированное программирование (ООП), обобщенное программирование и традиционное процедурное программирование. В пятом издании настоящей книги язык описывался как набор стандартов ISO C++, который неформально назывался C++99 и C++03 или иногда – C++99/03. (Версия 2003 была в основном формальным исправлением стандарта 1999 без добавления каких-либо новых возможностей.) С тех пор язык C++ продолжал развиваться. На момент написания данной книги международный комитет по стандартам C++ одобрил новую версию стандарта. Во время разработки этот стандарт имел неформальное название C++0x, а теперь он будет известен как C++11. Большинство современных компиляторов поддерживают C++99/03 достаточно хорошо, и многие примеры в этой книги соответствуют этому стандарту. Однако в некоторых реализациях уже появились многие возможности, описанные новым стандартом, и эти возможности рассматриваются в настоящем издании книги.

В этой книге обсуждается базовый язык C и текущие функциональные средства C++, что делает ее самодостаточной. В ней представлены основы языка C++, иллюстрируемые с помощью коротких и точных программ, которые легко скопировать для дальнейших экспериментов. Вы узнаете о вводе-выводе, о решении повторяющихся задач и возможностях выбора, о способах обработки данных и о функциях. Будут описаны многие средства C++, которые были добавлены к языку C, включая перечисленные ниже:

- классы и объекты;
- наследование;
- полиморфизм, виртуальные функции и идентификация типов во время выполнения (RTTI);
- перегрузка функций;
- ссылочные переменные;
- обобщенное (или не зависящее от типов) программирование, обеспечиваемое шаблонами и стандартной библиотекой шаблонов (STL);
- механизм исключений для обработки ошибочных условий;
- пространства имен для управления именами функций, классов и переменных.

Принятый подход

Методика изложения материала в этом учебнике обладает рядом преимуществ. Книга написана в духе традиций, которые сложились почти двадцать лет назад с момента выхода книги *Язык программирования C. Лекции и упражнения*, и воплощает в себе следующие принципы.

- Учебник должен быть легко читаемым и удобным в качестве руководства.
- Предполагается, что читатель еще не знаком с принципами программирования, имеющими отношение к теме учебника.
- Изложение материала сопровождается небольшими простыми примерами, которые читатель может выполнить самостоятельно. Примеры способствуют пониманию изложенного материала.

- Концепции в учебнике проясняются с помощью иллюстраций.
- Учебник должен содержать вопросы и упражнения, которые позволят читателю закрепить пройденный материал. Благодаря этому, учебник можно использовать как для самостоятельного, так и для коллективного обучения.

Книга, написанная в соответствии с этими правилами, поможет разобраться во всех тонкостях языка программирования C++ и научит использовать его для решения конкретных задач.

- В учебнике содержится концептуальное руководство по использованию конкретных функциональных средств, таких как применение открытого наследования для моделирования отношений *является*.
- В учебнике приведены примеры распространенных стилей и приемов программирования на языке C++.
- В учебнике имеется большое количество врезок, в том числе советы, предостережения, памятки и примеры из практики.

Автор и редакторы этого учебника приложили максимум усилий, чтобы он получился простым, материал был понятным, и вы остались довольными прочитанным. Мы стремились к тому, чтобы после изучения предложенного материала вы смогли самостоятельно писать надежные и эффективные программы и получать удовольствие от этого занятия.

Примеры кода, используемые в книге

Учебник изобилует примерами кода, большинство из которых являются завершенными программами. Как и в предыдущих изданиях, примеры написаны на обобщенном языке C++, поэтому ни один из них не привязан к определенному типу компьютера, операционной системе или компилятору. Примеры тестировались в системах Windows 7, Macintosh OS X и Linux. В некоторых программах используются средства C++11, и они требуют компиляторов, поддерживающих эти средства, но остальные программы должны работать в любой системе, совместимой с C++99/03.

Код завершенных программ, рассмотренных в этой книге, доступен для загрузки на веб-сайте издательства.

Организация книги

Эта книга содержит 18 глав и 10 приложений, которые кратко описаны ниже.

- **Глава 1, "Начало работы с C++".** В главе 1 рассказывается о том, как Бьярне Страуструп создал язык программирования C++, добавив к языку C поддержку объектно-ориентированного программирования. Вы узнаете об отличиях между процедурными языками программирования, такими как C, и объектно-ориентированными языками, примером которых является C++. Вы узнаете, как комитетами ANSI/ISO был разработан и утвержден стандарт C++. В главе также рассматривается порядок создания программы на C++ с учетом особенностей множества современных компиляторов C++. В конце главы приведены соглашения, используемые в этой книге.
- **Глава 2, "Приступаем к изучению C++".** В главе 2 подробно объясняется процесс написания простых программ на C++. Вы узнаете о роли функции `main()` и о некоторых разновидностях операторов, используемых в программах C++. Для операций ввода-вывода в программах вы будете использовать предопределенные объек-

ты cout и cin, а еще вы научитесь создавать и использовать переменные. В конце главы вы познакомитесь с функциями — программными модулями языка C++.

- **Глава 3, "Работа с данными".** Язык программирования C++ предоставляет встроенные типы для хранения двух разновидностей данных — целых чисел (чисел без дробной части) и чисел с плавающей точкой (чисел с дробной частью). Чтобы удовлетворить разнообразные требования программистов, для каждой категории данных в C++ предлагается несколько типов. В главе 3 рассматриваются как сами эти типы, так и создание переменных и написание констант различных типов. Вы также узнаете о том, как C++ обрабатывает явные и неявные преобразования одного типа в другой.
- **Глава 4, "Составные типы".** Язык C++ позволяет строить на основе базовых встроенных типов данных более развитые типы. Самой сложной формой являются классы, о которых пойдет речь в главах 9–13. В главе 4 рассматриваются другие формы, включая массивы, которые хранят множество значений одного и того же типа; структуры, хранящие несколько значений разных типов; и указатели, которые идентифицируют ячейки памяти. Вы узнаете о том, как создавать и хранить текстовые строки и как обрабатывать текстовый ввод-вывод за счет использования символьных массивов с стиле C и класса string из C++. Наконец, будут описаны некоторые способы обработки выделения памяти в C++, в том числе применение операций new и delete для явного управления памятью.
- **Глава 5, "Циклы и выражения отношений".** Программы часто должны выполнять повторяющиеся действия, и для этих целей в C++ предусмотрены три циклических структуры: цикл for, цикл while и цикл do while. Такие циклы должны знать, когда им необходимо завершаться, и операции отношения C++ позволяют создавать соответствующие проверочные выражения. В главе 5 вы узнаете, как создавать циклы, которые читают входные данные и обрабатывают их символ за символом. Наконец, вы научитесь создавать двумерные массивы и применять вложенные циклы для их обработки.
- **Глава 6, "Операторы ветвления и логические операции".** Поведение программы будет интеллектуальным, если она сможет адаптироваться к различным ситуациям. В главе 6 рассказывается об управлении ходом выполнения программы с помощью операторов if, if else и switch, а также условных операций. Вы узнаете, как использовать условные операции для проверки с целью принятия решений. Кроме этого, вы ознакомитесь с библиотекой функций cctype, которая предназначена для оценки символьных отношений, таких как проверка, является ли данный символ цифрой или непечатаемым символом. В конце главы дается краткий обзор файлового ввода-вывода.
- **Глава 7, "Функции как программные модули C++".** Функции являются базовыми строительными блоками в программировании на языке C++. Основное внимание в главе 7 сосредоточено на возможностях, которые функции C++ разделяют с функциями C. В частности, будет представлен общий формат определения функций и показано, как с помощью прототипов функций можно увеличить надежность программ. Вы научитесь создавать функции для обработки массивов, символьных строк и структур. Кроме того, вы узнаете о рекурсии, которая возникает, когда функция обращается к самой себе, а также о том, как с помощью функции можно реализовать стратегию "разделяй и властвуй". Наконец, вы познакомитесь с указателями на функции, благодаря которым одна функция может с помощью аргумента пользоваться другой функцией.

- **Глава 8, "Дополнительные сведения о функциях".** В главе 8 будет рассказано о новых средствах C++, доступных для функций. Вы узнаете, что такое встроенные функции, с помощью которых можно ускорить выполнение программы за счет увеличения ее размера. Вы будете работать со ссылочными переменными, которые предлагают альтернативный способ передачи информации функциям. Аргументы по умолчанию позволяют функции автоматически задавать значения тем аргументам, которые при вызове функции не указаны. Перегрузка функций позволяет создавать функции, которые имеют одно и то же имя, но принимают разные списки аргументов. Все эти средства часто используются при проектировании классов. Вы узнаете также о шаблонах функций, благодаря которым можно строить целые семейства связанных функций.
- **Глава 9, "Модели памяти и пространства имен".** В главе 9 рассматриваются вопросы построения программ, состоящих из множества файлов. В ней будут представлены варианты выделения памяти и методы управления памятью; вы узнаете, что такое область видимости, связывание и пространства имен, которые определяют, каким частям программы будут известны та или иная переменная.
- **Глава 10, "Объекты и классы".** Класс представляет собой определяемый пользователем тип, а объект (такой как переменная) является экземпляром класса. В главе 10 вы узнаете о том, что такое объектно-ориентированное программирование, и как проектируются классы. Объявление класса описывает информацию, хранящуюся в объекте класса, и операции (методы класса), разрешенные для объектов класса. Некоторые части объекта будут видимыми внешнему миру (открытая часть), а другие – скрытыми (закрытая часть). В момент создания и уничтожения объектов инициируются специальные методы класса, называемые конструкторами и деструкторами. В этой главе вы узнаете об этих и других особенностях классов и получите представление об использовании классов для реализации абстрактных типов данных, таких как стек.
- **Глава 11, "Работа с классами".** В главе 11 продолжается знакомство с классами. Прежде всего, будет рассмотрен механизм перегрузки операций, который позволяет определять, каким образом операции вроде + будут работать с объектами класса. Вы узнаете о дружественных функциях, которые могут получать доступ к данным класса, в общем случае недоступных извне. Будет показано, как некоторые конструкторы и функции-члены перегруженных операций могут быть использованы для управления преобразованием одного типа класса в другой.
- **Глава 12, "Классы и динамическое выделение памяти".** Часто требуется, чтобы член класса указывал на динамически выделенную память. Если в конструкторе класса используется операция `new` для динамического выделения памяти, то должен быть предусмотрен соответствующий деструктор, явный конструктор копирования и явная операция присваивания. В главе 12 будет показано, как это можно сделать, и описано поведение функций-членов, которые генерируются неявным образом, если явные описания не предоставлены. Опыт работы с классами расширяется на использование указателей на объекты и моделирование очередей.
- **Глава 13, "Наследование классов".** Одним из самых мощных средств объектно-ориентированного программирования является наследование, благодаря которому производный класс наследует возможности базового класса, позволяя повторно использовать код базового класса. В главе 13 обсуждается открытое наследование, моделирующее отношения *является*, при которых производный

объект рассматривается как специальный случай базового объекта. Например, физик является специальным случаем ученого. Некоторые отношения наследования являются полиморфными, что означает возможность писать код, используя несколько связанных между собой классов, для которых метод с одним и тем же именем может реализовывать поведение, зависящее от типа объекта. Для реализации такого поведения необходимо применять новый вид функции-члена – виртуальную функцию. Иногда для отношений наследования лучше всего использовать абстрактные базовые классы. В этой главе будут рассмотрены все эти вопросы с описанием ситуаций, в которых открытое наследование имеет смысл.

- **Глава 14, "Повторное использование кода в C++".** Открытое наследование – это всего лишь один из способов повторного использования кода. В главе 14 рассматриваются другие варианты. Ситуация, при которой члены одного класса являются объектами другого класса, называется включением. Включение можно использовать для моделирования отношений *содержит*, при которых один класс имеет компоненты, принадлежащие другому классу (например, автомобиль содержит двигатель). Для моделирования таких отношений можно использовать закрытое и защищенное наследование. Вы узнаете о каждом способе и об отличиях разных подходов. Кроме того, будут описаны шаблоны классов, которые позволяют определить класс посредством некоторого неуказанного обобщенного типа, а затем использовать шаблон для создания специфических классов в терминах определенных типов. Например, с помощью шаблона стека можно создать стек целых чисел и стек строк. Наконец, вы узнаете о множественном открытом наследовании, при котором один класс может быть произведен из более чем одного класса.
- **Глава 15, "Друзья, исключения и многое другое".** В главе 15 речь пойдет о так называемых "друзьях", а именно – о дружественных классах и дружественных функциях-членах. Здесь вы найдете сведения о некоторых нововведениях C++, начиная с исключений, которые предлагают механизм обработки необычных ситуаций, возникающих при выполнении программы (например, неподходящие значения аргументов функции или нехватка памяти). В этой главе вы узнаете, что собой представляет механизм RTTI (идентификация типов во время выполнения). В заключительной части главы будет рассказано о безопасных альтернативах неограниченному приведению типов.
- **Глава 16, "Класс `string` и стандартная библиотека шаблонов".** В главе 16 рассказывается о некоторых полезных библиотеках классов, недавно добавленных к языку. Класс `string` является удобной и мощной альтернативой традиционным строкам в стиле C. Класс `auto_ptr` помогает управлять динамически выделяемой памятью. Библиотека STL содержит множество обобщенных контейнеров, включая шаблонные представления массивов, очередей, списков, множеств и карт. Она также предоставляет эффективную библиотеку обобщенных алгоритмов, которые можно использовать с контейнерами STL и обычными массивами. Шаблонный класс `valarray` обеспечивает поддержку для числовых массивов.
- **Глава 17, "Ввод, вывод и файлы".** В главе 17 рассматривается ввод-вывод в C++ и обсуждаются вопросы форматирования вывода. Вы узнаете, как использовать методы классов для определения состояния потоков ввода и вывода, чтобы проверить, например, присутствует во входных данных несоответствие типов либо условие достижения конца файла. Чтобы произвести классы для управле-

ния файловым вводом и выводом, в C++ используется наследование. В главе будет показано, как открывать файлы для ввода и вывода, как добавлять данные в файл, как работать с бинарными файлами и как получить произвольный доступ к файлу. Напоследок вы узнаете о том, как применять стандартные методы ввода-вывода для чтения и записи строк.

- **Глава 18, "Новый стандарт C++".** Глава 18 начинается с обзора множества функциональных средств C++11, которые были введены в предшествующих главах, в том числе новые типы, унифицированный синтаксис инициализации, автоматическое выведение типов, новые интеллектуальные указатели и перечисления с областью видимости. Затем в главе обсуждается новый тип ссылки rvalue и показано, как он используется для реализации нового средства под названием *семантика переноса*. Кроме того, рассматриваются новые возможности классов, лямбда-выражения и шаблоны с переменным числом аргументов. Наконец, в главе даны описания многих новых средств, которые не упоминались в предшествующих главах книги.
- **Приложение А, "Основания систем счисления".** В приложении А обсуждаются восьмеричные, шестнадцатеричные и двоичные числа.
- **Приложение Б, "Зарезервированные слова C++".** В приложении Б приведен список ключевых слов C++.
- **Приложение В, "Набор символов ASCII".** В приложении В приведен набор символов ASCII вместе с его десятичным, восьмеричным, шестнадцатеричным и двоичным представлениями.
- **Приложение Г, "Приоритеты операций".** В приложении Г перечислены операции C++ в порядке убывания приоритетов.
- **Приложение Д, "Другие операции".** В приложении Д приводятся сведения об операциях C++, которые не были рассмотрены в основном тексте (например, поразрядные операции).
- **Приложение Е, "Шаблонный класс `string`".** В приложении Е приводятся сведения о методах и функциях класса `string`.
- **Приложение Ж, "Методы и функции стандартной библиотеки шаблонов".** В приложении Ж приводятся сведения о методах контейнеров STL и функциях общих алгоритмов STL.
- **Приложение З, "Рекомендуемая литература и ресурсы в Интернете".** В приложении З предложен список книг и ресурсов, которые можно использовать для дальнейшего изучения C++.
- **Приложение И, "Переход к стандарту ANSI/ISO C++".** В приложении И представлено руководство по переносу кода на C и старых реализаций C++ в код на ANSI/ISO C++.
- **Приложение К, "Ответы на вопросы для самоконтроля".** В приложении К содержатся ответы на вопросы для самоконтроля, приведенные в конце каждой главы.

Примечание для преподавателей

Одна из целей этого издания заключалась в том, чтобы предложить книгу, которую можно было бы использовать либо в качестве самоучителя, либо в качестве учебника.

Ниже перечислены некоторые характеристики, которые позволяют рассматривать эту книгу как учебник.

- Книга описывает обобщенный язык программирования C++, т.е. не зависит от конкретной реализации.
- Предложенный материал соответствует стандарту ANSI/ISO C++ и включает обсуждение шаблонов, библиотеки STL, класса `string`, исключений, RTTI и пространств имен.
- Для чтения книги не требуются предварительные знания языка C (хотя определенные навыки в программировании желательны).
- Темы упорядочены таким образом, что первые главы могут быть изучены довольно быстро как обзорные главы на курсах, слушатели которых знакомы с языком C.
- В конце каждой главы предлагаются вопросы для самоконтроля и упражнения по программированию. В приложении К даны ответы на вопросы для самоконтроля.
- В книге присутствуют темы, подходящие для курсов по вычислительной технике, включая абстрактные типы данных, стеки, очереди, простые списки, эмуляции, обобщенное программирование и применение рекурсии для реализации стратегии "разделяй и властвуй".
- Большинство глав имеют такой объем, чтобы их можно было изучить в течение одной недели или даже быстрее.
- В книге обсуждается, *когда* можно использовать определенные функциональные средства, и *каким образом* их использовать. Например, открытое наследование связывается с отношениями *содержит*, композиция и закрытое наследование – с отношениями *является*, кроме того, рассматриваются вопросы о том, когда использовать виртуальные функции, а когда – нет.

Соглашения, используемые в этой книге

Для выделения различных частей текста в книге используются следующие типографские соглашения.

- Строки кода, команды, операторы, переменные, имена файлов и вывод программ выделяется моноширинным шрифтом:

```
#include <iostream>
int main()
{
   using namespace std;
   cout << "What's up, Doc!\n";
   return 0;
}
```

- Входные данные для программ, которые вы должны вводить самостоятельно, представлены **моноширинным шрифтом с полужирным начертанием**:

```
Please enter your name:
Plato
```

- Заполнители в описаниях синтаксиса выделены *моноширинным шрифтом с курсивным начертанием*. Каждый такой заполнитель должен заменяться реальным именем файла, параметром или любым другим элементом, который он представляет.
- Для выделения новых терминов используется *курсив*.

Врезка

Врезка содержит более детальное обсуждение или дополнительные сведения по текущему вопросу.

Совет

Здесь вы найдете краткие полезные советы и рекомендации по программированию.

Внимание!

Здесь вы встретите предупреждения о потенциальных ловушках.

На заметку!

Здесь предоставляются разнообразные комментарии, которые не подпадают ни под одну из указанных выше категорий.

Системы, на которых разрабатывались примеры для книги

Примеры на C++11 в этой книге разрабатывались с использованием Microsoft Visual C++ 2010 и Cygwin с Gnu g++ 4.5.0 в среде 64-разрядной ОС Windows 7. Остальные примеры тестировались в этих системах, а также в системе iMac с применением g++ 4.2.1 под OS X 10.6.8 и в системе Ubuntu Linux с использованием g++ 4.4.1. Большинство примеров, предшествующих C++11, изначально разрабатывались с помощью Microsoft Visual C++ 2003 и Metrowerks CodeWarrior Development Studio 9 в среде Windows XP Professional и тестировались с применением компиляторов командной строки Borland C++ 5.5 и GNU gpp 3.3.3 в той же системе, компиляторов Comeau 4.3.3 и GNU g++ 3.3.1 в системе SuSE 9.0 Linux и Metrowerks Development Studio 9 на Macintosh G4 под OS 10.3.

Для любого программиста язык C++ открывает целый мир возможностей; изучайте его и наслаждайтесь работой!

1

Начало работы с C++

В ЭТОЙ ГЛАВЕ...

- История и философия развития языков C и C++
- Сравнение процедурного и объектно-ориентированного программирования
- Добавление принципов объектно-ориентированного программирования в язык C
- Добавление принципов обобщенного программирования в язык C
- Стандарты языков программирования
- Порядок создания программы

Добро пожаловать в C++! Этот удивительный язык, сочетающий в себе функциональные возможности языка C и принципы объектно-ориентированного и обобщенного программирования, в девяностых годах прошлого столетия занял лидирующую позицию среди существующих языков программирования, продолжая удерживать ее и в двухтысячных годах. Своим происхождением он обязан языку программирования C, поэтому ему свойственны такие характеристики, как эффективность, компактность, быстродействие и переносимость. Благодаря принципам объектной ориентации, язык программирования C++ предлагает новую методологию программирования, которая позволяет решать современные задачи, степень сложности которых постоянно растет. Благодаря возможности использования шаблонов, язык C++ предлагает еще одну новую методологию: обобщенное программирование. Во всем этом есть свои положительные и отрицательные стороны. В конечном счете, C++ является очень мощным языком программирования, но вместе с тем на его изучение нужно потратить довольно много времени.

Мы начнем с небольшого экскурса в историю происхождения языка C++, а затем перейдем к основным правилам создания программ на C++. Оставшаяся часть книги будет сконцентрирована на том, чтобы научить читателя программированию на языке C++: вначале мы рассмотрим простые основы языка, после чего приступим к изучению объектно-ориентированного программирования (ООП), освоим весь его жаргон — объекты, классы, инкапсуляцию, сокрытие данных, полиморфизм и наследование — и напоследок перейдем к изучению обобщенного программирования. (Естественно, по мере изучения C++ эти профессиональные термины перейдут из разряда модных словечек в лексикон опытного разработчика.)

Изучение языка C++: с чем придется иметь дело

В языке программирования C++ объединены три отдельных категории программирования: процедурный язык, представленный C; объектно-ориентированный язык, представленный расширениями в форме классов, которые C++ добавляет к C; и обобщенное программирование, поддерживаемое шаблонами C++. В этой главе как раз и рассматриваются упомянутые категории. Однако, прежде всего давайте разберемся с тем, как эти категории влияют на изучение C++. Одна из причин использования языка C++ связана с желанием получить в свое распоряжение возможности объектной ориентации. Для этого вы должны обладать навыками работы со стандартным языком C, который предлагает для C++ базовые типы, операции, управляющие структуры и синтаксические правила. Поэтому, зная язык программирования C, вы можете смело приступать к изучению C++. Это, однако, вовсе не означает, что вам достаточно будет изучить одни лишь ключевые слова и конструкции. Переход от C к C++ может потребовать стольких же усилий, сколько изучение языка C с самого начала. Кроме этого, если вы знакомы с языком C, то при переходе к C++ вы должны будете отказаться от привычного вам способа написания программ. Если вы не знаете C, то для изучения языка C++ вам придется освоить компоненты языка C, компоненты ООП и обобщенные компоненты, но, по крайней мере, вам не придется забывать привычный способ написания программ. Если у вас начинает складываться впечатление, что для изучения языка программирования C++ придется хорошенько напрячь свои умственные способности, то вы не ошибаетесь. Эта книга послужит хорошим помощником в процессе обучения, поэтому вы сможете сэкономить свои силы.

Настоящая книга построена таким образом, что в ней рассматриваются как элементы языка C, лежащего в основе C++, так и новые компоненты, поэтому она будет полезна даже для неподготовленных читателей. Процесс обучения начинается с рассмотрения функциональных возможностей, общих для обоих языков. Даже если вы знаете язык C, эта часть книги окажется полезным повторением. В ней уделено внимание концепциям, значение которых будет раскрыто позже, и показаны различия между C++ и C. После того как вы усвоите основы языка C, мы перейдем к знакомству с суперструктурой C++. С этого момента мы займемся рассмотрением объектов и классов и вы узнаете, как они реализованы в C++. Затем мы обратимся к шаблонам.

Эта книга не является полным справочником по языку C++, и в ней не рассматриваются все его тонкости. Однако у вас есть возможность изучить большинство основных особенностей языка, в том числе шаблоны, исключения и пространства имен.

А теперь давайте рассмотрим вкратце историю происхождения языка C++.

Истоки языка C++: немного истории

Развитие компьютерных технологий в течение последних нескольких десятков лет происходило удивительно быстрыми темпами. Современные ноутбуки могут производить вычисления гораздо быстрее и хранить намного больше информации, чем вычислительные машины 60-х годов прошлого столетия. (Очень немногие программисты могут вспомнить, как им приходилось возиться с колодами перфокарт, чтобы загрузить их в огромную компьютерную систему, занимающую отдельную комнату и имеющую немыслимый по тем временам объем памяти 100 Кбайт — намного меньше, чем в настоящее время располагает рядовой смартфон.) В ногу со временем развивались и языки программирования. Их развитие не было столь впечатляющим, однако имело огромное значение. Для больших и мощных компьютеров требовались более сложные программы, для которых, в свою очередь, нужно было решать новые задачи управления и сопровождения программ.

В семидесятых годах прошлого столетия такие языки программирования, как C и Pascal, способствовали зарождению эры структурного программирования, привнесшего столь необходимые на то время порядок и дисциплину. Помимо того, что язык C предлагал инструментальные средства для структурного программирования, с его помощью можно было создавать компактные быстро выполняющиеся программы, а также справляться с решением аппаратных задач, например, с управлением коммуникационными портами и дисковыми накопителями. Благодаря этим характеристикам, в восьмидесятых годах язык C стал доминирующим языком программирования. Наряду с ростом популярности языка C зарождалась и новая парадигма программирования: объектно-ориентированное программирование (ООП), которое было реализовано в таких языках, как SmallTalk и C++. Давайте рассмотрим взаимоотношения языка C и ООП более подробно.

Язык программирования C

В начале семидесятых годов прошлого столетия Деннис Ритчи (Dennis Ritchie), сотрудник компании Bell Laboratories, участвовал в проекте по разработке операционной системы Unix. (*Операционная система* (ОС) представляет собой набор программ, предназначенных для управления аппаратными ресурсами и обслуживания взаимодействий пользователя и компьютера. Так, например, именно ОС выводит на экран монитора системное приглашение в терминальном интерфейсе, управляет окнами и мышью в графическом интерфейсе и запускает программы на выполнение.) В своей

работе Ритчи нуждался в языке программирования, который был бы лаконичным, с помощью которого можно было бы создавать компактные и быстро выполняющиеся программы, и посредством которого можно было бы эффективно управлять аппаратными средствами.

По сложившейся на то время традиции программисты использовали для решения этих задач язык ассемблера, тесно связанный с внутренним машинным языком. Однако язык ассемблера является языком *низкого уровня*, т.е. он работает непосредственно с оборудованием (например, напрямую обращается к регистрам центрального процессора и ячейкам памяти). Таким образом, язык ассемблера является специфическим для каждого процессора компьютера. Поэтому если вы хотите, чтобы ассемблерная программа, написанная для компьютера одного типа, могла работать на компьютере другого типа, то, возможно, вам придется полностью переписать программу на другом языке ассемблера. Эта ситуация похожа на то, когда вы приобрели новый автомобиль и обнаружили, что конструкторы изменили расположение рычагов управления и их назначение, поэтому вам теперь нужно заново научиться водить машину.

Однако ОС Unix предназначалась для работы на самых разнообразных типах компьютеров (или платформ). Таким образом, это предполагало использование языка программирования *высокого уровня*. Такой язык ориентирован на решение задач, а не на обслуживание определенного оборудования. Специальные программы, называемые *компиляторами*, переводят язык программирования высокого уровня на внутренний язык определенного компьютера. Поэтому одну и ту же программу, написанную на языке программирования высокого уровня, можно запускать на различных платформах, применяя разные компиляторы. Ритчи необходим был язык, который сочетал бы в себе эффективность языка низкого уровня и возможность доступа к аппаратным средствам с универсальностью и переносимостью языка высокого уровня. Поэтому, основываясь на старых языках программирования, он создал язык C.

Философия программирования на языке C

Поскольку C++ прививает языку C новые принципы программирования, мы должны сначала изучить философию программирования на языке C. В общем случае компьютерные языки имеют дело с двумя концепциями – данные и алгоритмы. *Данные* – это информация, которую использует и обрабатывает программа. *Алгоритмы* – это методы, используемые программой (рис. 1.1). Как и большинство распространенных в то время языков программирования, язык C был *процедурным* языком программирования. Это означает, что в этом языке акцент ставился на обработку данных с помощью алгоритмов. Суть процедурного программирования заключается в том, чтобы запланировать действия, которые должен предпринять компьютер, и с помощью языка программирования их реализовать. В программе предварительно описывается некоторое количество процедур, которые должен будет выполнить компьютер, чтобы получить определенный результат. Здесь в качестве примера можно упомянуть кулинарный рецепт, в котором записан порядок действий, выполнив которые кондитер сможет получить пирог.

В ранних процедурных языках программирования, к которым относятся FORTRAN и BASIC, с увеличением размера программ возникали организационные проблемы. Например, в программах часто используются операторы ветвления, которые передают выполнение тому или иному набору инструкций в зависимости от результата проверки.

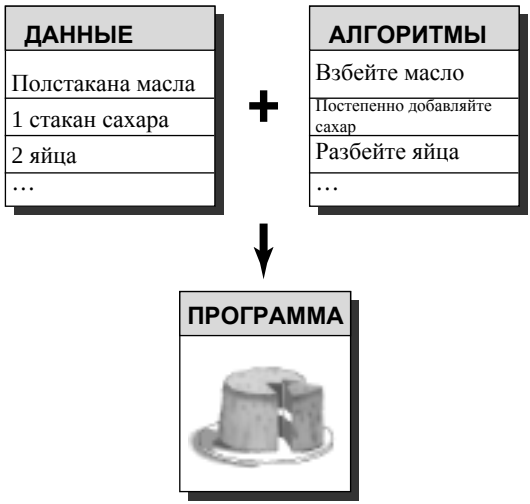


Рис. 1.1. *Данные + алгоритмы = программа*

В большинстве старых программ было настолько много запутанных переходов (на профессиональном жаргоне это называется "спагетти-кодом"), что при прочтении программу практически невозможно было понять, и попытка модифицировать такую программу сулила одни неприятности. Чтобы выйти из ситуации такого рода, специалисты по вычислительной технике разработали более дисциплинированный стиль программирования, называемый *структурным программированием*.

Язык программирования C обладает всеми возможностями для реализации этого подхода. Например, структурное программирование ограничивает ветвление (выбор следующей инструкции для выполнения) небольшим набором удобных и гибких конструкций. В языке C эти конструкции (циклы `for`, `while`, `do while` и оператор `if else`) включены в его словарь.

Другим новшеством было так называемое *нисходящее проектирование*. В языке C эта идея состояла в том, чтобы разделить большую программу на небольшие, поддающиеся управлению задачи. Если после разбиения одна из задач все равно остается крупной, этот процесс продолжается до тех пор, пока программа не будет разделена на небольшие модули, с которыми будет просто работать. (Организуйте свой процесс обучения. Ах, нет! Приведите в порядок свой рабочий стол, картотеку и наведите порядок на книжных полках. Ах, нет! Начните со стола, наведите порядок в каждом выдвижном ящике, начиная со среднего. Да, судя по всему, справиться с этой задачей вполне реально.) Этот подход вполне осуществим в языке C, т.к. он позволяет разрабатывать программные модули, называемые *функциями*, которые отвечают за выполнение конкретной задачи. Как вы могли заметить, в структурном программировании отображено процедурное программирования, в том смысле, что программа представляется в виде действий, которые она должна выполнить.

Переход к C++: объектно-ориентированное программирование

Несмотря на то что принципы структурного программирования делают сопровождение программ более ясным, надежным и простым, написать большую программу все еще было непросто. Новый подход к решению этой проблемы был воплощен в

объектно-ориентированном программировании (ООП). В отличие от процедурного программирования, в котором акцент делается на алгоритмах, в ООП во главу угла поставлены данные. Вместо того чтобы пытаться решать задачу, приспосабливая ее к процедурному подходу в языке программирования, в ООП язык программирования приспосабливается к решению задачи. Суть заключается в том, чтобы создать такие формы данных, которые могли бы выразить важные характеристики решаемой задачи.

В языке программирования C++ существуют понятия *класса*, который представляет собой спецификацию, описывающую такую новую форму данных, и *объекта*, который представляет собой индивидуальную структуру данных, созданную в соответствии с этой спецификацией. Например, класс может описывать общие свойства руководителя компании (фамилия, занимаемая должность, годовой доход, необычные способности и т.п.), а объект может представлять конкретного человека (например, Гилфорд Шипблатт, вице-президент компании, годовой доход составляет $925 000, умеет восстанавливать системный реестр Windows). В общем, класс описывает, какие данные используются для отображения объекта *и* какие операции могут быть выполнены над этими данными. Можно, например, определить класс для описания прямоугольника. Часть, касающаяся данных, этой спецификации может включать расположение вершин прямоугольника, высоту и ширину, цвет и стиль линии контура, а также цвет шаблона для закраски прямоугольника. Часть, касающаяся операций, этой спецификации может содержать методы для перемещения прямоугольника, изменения его размеров, вращения, изменения цвета и шаблонов, а также копирования прямоугольника в другое местоположение. Если позже вы воспользуетесь своей программой для рисования прямоугольника, она сможет создать объект в соответствии с его описанием. Объект будет содержать все значения данных, описывающие прямоугольник, и для изменения прямоугольника можно применять методы класса. Если вы нарисуете два прямоугольника, программа создаст два объекта, по одному для каждого прямоугольника.

Подход к проектированию программ, применяемый в ООП, заключается в том, чтобы сначала спроектировать классы, в точности представляющие все элементы, с которыми будет работать программа. Например, программа рисования может работать с классами, представляющими прямоугольники, линии, окружности, кисти, перья и т.п. В описаниях классов, как вы теперь уже знаете, указываются разрешенные операции для каждого класса, такие как перемещение окружности или вращение линии. Затем можно продолжить процесс разработки программы уже с помощью объектов для этих классов. Процесс перехода с нижнего уровня организации, например, с классов, до верхнего уровня – проектирования программы, называется *восходящим* программированием.

ООП позволяет не только связывать данные и методы в единственное определение класса. Например, ООП упрощает создание повторно используемого кода, что позволяет иногда сократить большой объем работы. Сокрытие информации позволяет предотвратить несанкционированный доступ к данным. Благодаря полиморфизму можно создавать множество описаний для операций и функций с программным контекстом, определяющим, какое описание используется. Благодаря наследованию, можно порождать новые классы на основе существующих классов. Как видите, ООП привносит множество новых идей и, в отличие от процедурного программирования, является другим принципом программирования. Вместо того чтобы сосредоточиваться на задачах, вы концентрируете внимание на концепциях представления. В некоторых случаях вместо нисходящего можно применять восходящий подход. Все эти вопросы, сопровождаемые простыми и понятными примерами, будут рассмотрены в данной книге.

Спроектировать полезный и надежный класс не так-то просто. К счастью, языки объектно-ориентированного программирования упрощают встраивание существующих классов в решаемые задачи. Поставщики программного обеспечения предлагают большое количество полезных библиотек классов, в том числе библиотеки, упрощающие написание программ для сред вроде Windows или Macintosh. Одно из замечательных преимуществ языка C++ заключается в том, что он позволяет без особых сложностей использовать повторно и адаптировать существующий надежный код.

C++ и обобщенное программирование

Еще одной парадигмой программирования, реализованной в языке C++, является обобщенное программирование. Как и ООП, обобщенное программирование направлено на упрощение повторного использования кода и на абстрагирование общих концепций. Однако в ООП акцент программирования ставится на данные, а в обобщенном программировании — на независимость от конкретного типа данных. И его цель тоже другая. ООП — это инструмент для управления большими проектами, тогда как обобщенное программирование предлагает инструменты для решения простых задач, подобных сортировке данных или слияния списков. Термин *обобщенный* относится к коду, тип которого является независимым. Данные в C++ могут быть представлены разными способами: в виде целых чисел, чисел с дробной частью, в виде символов, строк символов и в форме определяемых пользователем сложных структур нескольких типов. Например, если вам необходимо сортировать данные различных типов, то в общем случае для каждого типа потребовалось бы создавать отдельную функцию сортировки. При обобщенном программировании язык расширен, поэтому вы можете один раз написать функцию для обобщенного (т.е. не указанного) типа и применять ее для множества существующих типов. Для этого в языке C++ предусмотрены шаблоны.

Происхождение языка программирования C++

Как и C, язык C++ был создан в начале восьмидесятых годов прошлого столетия в Bell Laboratories, где работал Бьярне Страуструп (Bjarne Stroustrup). Вот что об этом говорит сам Страуструп: "C++ был создан главным образом потому, что мои друзья, да и я сам, не имели никакого желания писать программы на ассемблере, C или каком-нибудь языке программирования высокого уровня, существовавшем в то время. Задача заключалась в том, чтобы сделать процесс написания хороших программ простым и более приятным для каждого программиста".

Домашняя страница Бьярне Страуструпа

Бьярне Страуструп, создатель языка программирования C++, является автором нескольких широко известных справочных руководств — *The C++ Programming Language* и *The Design and Evolution of C++*. Его персональный веб-сайт, размещенный в AT&T Labs Research, должен быть в числе ваших основных закладок:

`www.research.att.com/~bs`

На этом сайте можно найти интересные исторические предпосылки возникновения C++, биографические материалы Бьярне Страуструпа и часто задаваемые вопросы по C++. Удивительно, но чаще всего Страуструпа спрашивают о том, как правильно произносится его имя и фамилия. Просмотрите раздел часто задаваемых вопросов на указанном веб-сайте и загрузите файл `.WAV`, чтобы услышать это самому!

Страуструп стремился больше к тому, чтобы сделать язык C++ полезным, а не внедрить какой-нибудь новый принцип или стиль программирования. Средства языка C++ в первую очередь должны удовлетворять потребностям программиста, а не быть воплощением красивой теории. За основу C++ Страуструп взял язык C, известный своей лаконичностью, пригодностью для системного программирования, широкой доступностью и тесной взаимосвязью с ОС Unix.

Принципы ООП были позаимствованы в языке моделирования Simula67. Страуструпу удалось реализовать в языке C принципы ООП и поддержку обобщенного программирования без существенного изменения языка. Поэтому язык программирования C++ в первом приближении можно рассматривать как расширенный набор C; в пользу этого свидетельствует тот факт, что любая допустимая программа на языке C – это также допустимая программа на C++. Правда, существуют некоторые незначительные отличия, но не более того. Программы, написанные на C++, могут пользоваться существующими библиотеками программного обеспечения для C. *Библиотека* представляет собой коллекцию программных модулей, которые могут быть вызваны из программы. В этих библиотеках предлагаются надежные и проверенные решения ко многим наиболее часто встречающимся задачам программирования, позволяя тем самым экономить ваши усилия и время. Распространение языка C++ стало возможным во многом благодаря библиотекам.

Название языка C++ происходит от операции инкремента (++) в языке C, которая увеличивает на единицу значение переменной. Таким образом, имя C++ в точности отражает расширенную версию языка C.

Компьютерная программа переводит практическую задачу в последовательность действий, которые должен выполнить компьютер. Благодаря принципам ООП, язык C++ может устанавливать связь с концепциями, составляющими задачу, а благодаря возможностям языка C, он может работать с оборудованием (рис. 1.2). Такое сочетание возможностей способствовало росту популярности языка C++. Процесс переключения с одного аспекта программы на другой можно представить себе как процесс переключения передач в автомобиле. (В действительности некоторые приверженцы чистоты ООП считают, что реализация принципов ООП в языке C сродни попытке научить летать поросенка.) Кроме того, поскольку C++ внедряет принципы ООП в язык C, их можно просто проигнорировать. Однако в этом случае вы утратите массу возможностей.

После того, как C++ действительно стал популярным языком программирования, Страуструп ввел шаблоны, открыв возможности для обобщенного программирования. И только после этого стало ясно, насколько удачным было использование шаблонов – их значение оказалось даже большим, чем реализация ООП, хотя кто-то может и не согласиться с этим. Факт объединения в языке C++ ООП, обобщенного программирования и более традиционного процедурного подхода свидетельствует о том, что в C++ утилитарный подход господствует над идеологическим, и это одна из причин популярности данного языка.

Переносимость и стандарты

Представьте, что у себя на работе вы написали удобную программу на языке C++ для старенького ПК Pentium, функционирующего под управлением ОС Windows 2000, но руководство решило заменить эту машину новым компьютером, на котором установлена другая ОС, скажем, Mac OS X или Linux, и другой процессор, такой как SPARC. Сможете ли вы запустить свою программу на новой платформе? Естественно, вам придется повторно скомпилировать программу, используя компилятор C++ для

новой платформы. А нужно ли что-нибудь изменять в уже написанном коде? Если программу можно перекомпилировать, ничего в ней не меняя, и без помех запустить, то такая программа называется *переносимой*.

Рис. 1.2. *Двойственность языка C++*

Чтобы сделать программу переносимой, нужно справиться с двумя проблемами. Первая — это оборудование. Программа, настроенная на работу с конкретным аппаратным обеспечением, вряд ли будет переносимой. Если программа напрямую управляет платой видеоадаптера IBM PC, то на платформе Sun, например, она выдаст сплошную тарабарщину. (Проблему переносимости можно свести к минимуму, если локализовать части программы, зависящие от оборудования, в модулях функций; затем эти модули можно будет просто переписать.) В этой книге мы не будем рассматривать такой способ программирования.

Второй проблемой является несоответствие языков программирования. Вы наверняка согласитесь с утверждением, что в реальном мире тоже существует проблема с разговорными языками. Жители Бруклина, например, могут не понять сводку новостей на диалекте Йоркшира, и это притом, что все они разговаривают на английском языке. Такая же ситуация и в мире компьютеров: среди языков программирования можно различить характерные "диалекты". Хотя большинство разработчиков хотели бы добиться совместимости их собственных версий C++ с другими версиями, сделать это очень трудно без опубликованного стандарта, описывающего точную работу языка. В Национальном институте стандартизации США (American National Standards Institute — ANSI) в 1990 г. был сформирован комитет (ANSI X3J16), задача которого заключалась в разработке стандарта для языка программирования C++. (Стандарт для языка C уже был создан ANSI.)

Вскоре к этому процессу подключилась и Международная организация по стандартизации (International Organization for Standardization — ISO), у которой на тот момент был сформирован собственный комитет (ISO-WG-21). В результате усилия комитетов ANSI и ISO были объединены, после чего работа по созданию стандарта для языка C++ велась совместно.

Несколько лет напряженной работы, наконец, вылились в международный стандарт (ISO/IEC 14882:1998), который в 1998 г. был принят ISO, Международной электротехнической комиссией (International Electrotechnical Commission — IEC) и ANSI. Этот стандарт, часто называемый C++98, не только уточнял описание существующих средств языка C++, но также и расширений языка: исключений, идентификации типов во время выполнения (Runtime Type Identification — RTTI), шаблонов и стандартной библиотеки шаблонов (Standard Template Library — STL). В 2003 г. было опубликовано второе издание стандарта C++ (ISO/IEC 14882:2003); новое издание представляло собой формально пересмотренную версию. В ней были исправлены ошибки первого издания (ликвидированы опечатки, устранены неточности и т.п.), при этом средства языка программирования не менялись. Это издание стандарта часто называют C++03. Поскольку в C++03 средства самого языка остались не поменялись, мы будем понимать под C++98 как собственно C++98, так и C++03.

Язык C++ продолжает развиваться, и в августе 2011 г. комитет ISO одобрил новый стандарт под названием ISO/IEC 14882:2011, на который неформально ссылаются, как на C++11. Подобно C++98, стандарт C++11 добавляет к языку множество средств. Кроме того, его целями являются удаление противоречий, а также упрощение изучения и применения C++. Это стандарт был назван C++0x, при этом изначально ожидалось, что x будет 7 или 8, однако работа над стандартами — медленный, обстоятельный и утомительный процесс. К счастью, вскоре стало понятно, что 0x может рассматриваться как шестнадцатеричное целое число (см. приложение A), а это означало, что у комитета есть время до 2015 г. на завершение работы. Таким образом, согласно этому измерению, они закончили с опережением графика.

В стандарте ISO для языка C++ дополнительно приводится стандарт ANSI для языка C, поскольку считается, что C++ является расширенным набором C. Это означает, что любая допустимая программа на C в идеале также должна быть допустимой программой на C++. Между ANSI C и соответствующими правилами для C++ имеются некоторые различия, однако все они несущественны. Так, например, ANSI C включает некоторые возможности, впервые представленные в C++: прототипирование функций и спецификатор типа `const`.

До выхода ANSI C сообщество программистов на C следовало стандарту де-факто, основанному на книге *Язык программирования C* (Издательский дом "Вильямс", 2005 г.), написанной Брайаном Керниганом и Деннисом Ритчи. Этот стандарт часто назывался как K&R C; после появления ANSI C более простой стандарт K&R C теперь иногда называют *классическим C*.

Стандарт ANSI C не только определяет язык C, но и стандартную библиотеку C, которую должны поддерживать реализации ANSI C. Эта библиотека используется также и в C++; в книге мы называем ее *стандартной библиотекой C* или просто *стандартной библиотекой*. Кроме того, стандарт ISO C++ предоставляет стандартную библиотеку классов C++.

Стандарт C был пересмотрен и в результате получен стандарт C99, который был принят ISO в 1999 г. и ANSI в 2000 г. Этот стандарт добавил к языку C ряд средств, таких как новый целочисленный тип, который поддерживается некоторыми компиляторами C++.

Развитие языка

Первоначально стандарт де-факто для C++ был 65-страничным справочным руководством, включенным в 328-страничную книгу Страуструпа *The C++ Programming Language* (Addison-Wesley, 1986 г.).

Следующим важным опубликованным стандартом де-факто была книга Эллиса и Страуструпа *The Annotated C++ Reference Manual* (Addison-Wesley, 1990 г.). Ее объем составлял 453 страницы; в дополнение к справочному материалу она включала существенные комментарии.

Стандарт C++98 с добавленным множеством средств занимает около 800 страниц, при минимальном числе комментариев.

Описание стандарта C++11 потребовало свыше 1 350 страниц, что существенно превышает объем старого стандарта.

Эта книга и стандарты C++

Современные компиляторы обеспечивают хорошую поддержку для C++98. На момент написания этой книги некоторые компиляторы также поддерживали ряд средств C++11, и можно ожидать, что после принятия нового стандарта уровень поддержки станет быстро возрастать. Эта книга отражает текущую ситуацию, подробно раскрывая C++98 и представляя многие возможности C++11. Некоторые из этих возможностей описываются в рамках соответствующих тем по C++98. В главе 18 все внимание сосредоточено полностью на новых средствах, с подведением итогов по упомянутым ранее возможностям и представлением дополнительных новых средств.

Из-за неполной поддержки на время написания этой книги очень трудно адекватно представить абсолютно все новые возможности C++11. Но даже когда новый стандарт станет поддерживаться полностью, очевидно, что полное его изложение будет выходить далеко за рамки книги разумного объема. Подход, применяемый в настоящей книге, предусматривает концентрацию на средствах, которые уже доступны в ряде компиляторов, и краткое описание множества других возможностей.

Прежде чем вплотную заняться изучением языка C++, давайте рассмотрим порядок создания программ.

Порядок создания программы

Предположим, что вы написали программу на C++. Как вы собираетесь ее запускать? Действия, которые должны быть предприняты, зависят от компьютерной среды и применяемого компилятора C++, однако в общем случае они должны быть примерно следующими (рис. 1.3).

1. С помощью текстового редактора напишите свою программу и сохраните ее в файле. Это будет *исходный код* вашей программы.
2. Скомпилируйте исходный код. Для этого необходимо запустить программу, которая транслирует исходный код во внутренний язык, называемый *машинным языком* рабочего компьютера. Файл, содержащий транслированную программу — это *объектный код* вашей программы.

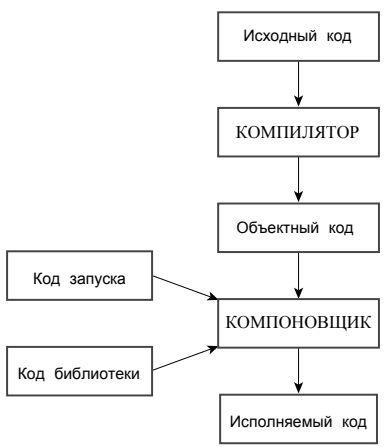


Рис. 1.3. Этапы программирования

3. Свяжите объектный код с дополнительным кодом. Например, программы на C++ обычно используют *библиотеки*. Библиотека C++ содержит объектный код для набора компьютерных подпрограмм, называемых *функциями*, которые решают такие задачи, как отображение информации на экране монитора, нахождение квадратного корня числа и т.д. В процессе связывания ваш объектный код комбинируется с объектным кодом для используемых функций и с некоторым стандартным кодом запуска для формирования версии времени выполнения вашей программы. Файл, содержащий этот финальный продукт, называется *исполняемым кодом*.

Термин *исходный код* вы будете постоянно встречать на страницах этой книги, поэтому постарайтесь запомнить его.

Большинство программ в этой книге являются обобщенными и должны выполняться в любой системе, которая поддерживает C++98. Однако некоторые программы, в частности, рассматриваемые в главе 18, требуют определенной поддержки C++11. На момент написания этой книги некоторые компиляторы требовали указания дополнительных флагов для активизации их частичной поддержки C++11. Например, компилятор `g++`, начиная с версии 4.3, в настоящее время при компиляции файла исходного кода использует флаг `-std=c++11`:

```
g++ -std=c++11 use_auto.cpp
```

Этапы сборки программы в одно целое могут варьироваться. Давайте рассмотрим их более подробно.

Создание файла исходного кода

В оставшейся части этой книги мы будем говорить обо всем, что связано с файлом исходного кода; в этом разделе речь пойдет о механизме создания этого файла. В некоторых реализациях языка C++, таких как Microsoft Visual C++, Embarcadero C++ Builder, Apple Xcode, Open Watcom C++, Digital Mars C++ и Freescale CodeWarrior, предлагается так называемая *интегрированная среда разработки* (Integrated Development Environment – IDE), которая позволяет управлять всеми этапами создания программы, включая редактирование, из одной главной программы. В других реализациях C++, таких как GNU C++ на платформах Unix и Linux, IBM XL C/C++ на платформе AIX, а также в свободно распространяемых версиях компиляторов Borland 5.5 (распространяемого Embarcadero) и Digital Mars, поддерживаются только этапы компиляции и компоновки, и все команды должны вводиться в командной строке. В этих случаях для создания и изменения исходного кода можно использовать любой доступный текстовый редактор. В системе Unix можно применять редакторы `vi`, `ed` или `emacs`. В режиме командной строки системы Windows можно работать в `edlin` либо `edit` или любом другом доступном текстовом редакторе. Можно даже использовать текстовый процессор при условии, что искомый файл будет сохраняться в формате текстового файла ASCII, а не в специальном формате текстового процессора. В качестве альтернативы может быть доступны IDE-среды для работы с упомянутыми компиляторами командной строки.

При назначении имени файлу исходного кода должен использоваться подходящий суффикс, с помощью которого файл можно идентифицировать как файл кода C++. Благодаря этому суффиксу не только вы, но и компилятор будет знать, что данный файл содержит исходный код C++. (Если Unix-компилятор пожалуется на неверное магическое число ("bad magic number"), следует иметь в виду, что это его излюбленный способ указывать на неверный суффикс.) Суффикс начинается с точки, за которой следует символ или группа символов, называемых *расширением* (рис. 1.4).

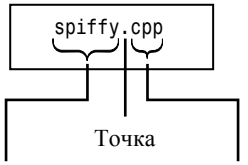


***Рис. 1.4.** Элементы имени файла исходного кода*

Выбор расширения будет зависеть от реализации C++. В табл. 1.1 перечислены некоторые распространенные варианты. Например, `spiffy.C` является допустимым именем Unix-файла исходного кода. Обратите внимание, что в Unix важно соблюдать регистр символов: символ `C` должен быть записан в верхнем регистре. Следует отметить, что расширения, записанные в нижнем регистре, тоже допускаются, однако в стандартной версии языка C используется именно такой формат расширения. Поэтому во избежание путаницы в Unix-системах следует применять `c` для программ на языке C и `C` — для программ на языке C++. Можно также использовать один или два дополнительных символа: в некоторых Unix-системах, например, можно использовать расширения `cc` и `cxx`. В DOS, более простой системе по сравнению с Unix, нет разницы в том, в каком регистре будет введен символ, поэтому для различия программ на C и C++ в DOS-реализациях применяются дополнительные символы (см. табл. 1.1).

Таблица 1.1. Расширения файла исходного кода

Реализация C++	Расширения файла исходного кода
Unix	`C, cc, cxx, c`
GNU C++	`C, cc, cxx, cpp, c++`
Digital Mars	`cpp, cxx`
Borland C++	`cpp`
Watcom	`cpp`
Microsoft Visual C++	`cpp, cxx, cc`
Freestyle CodeWarrior	`cpp, cp, cc, cxx, c++`

Компиляция и компоновка

Первоначально Страуструп реализовал C++ с программой компилятора из C++ в C, и не стал разрабатывать прямой компилятор из C++ в объектный код. Эта программа, называемая `cfront` (сокращение от *C front end*), транслировала исходный код C++ в исходный код C, который затем можно было компилировать с помощью стандартного компилятора C. Этот подход способствовал росту популярности C++ в среде программистов на C. В других реализациях этот подход использовался для переноса C++ на другие платформы. По мере совершенствования языка C++ и роста его популярности, все большее число разработчиков предпочитали создавать компиляторы C++, которые генерировали объектный код непосредственно из исходного кода C++. Такой прямой подход ускорял процесс компиляции и обособлял язык C++.

Способ компиляции зависит от реализации, и в последующих разделах мы рассмотрим некоторые варианты. В этих разделах будут описаны только основные этапы, однако это вовсе не означает, что вам не следует изучать документацию по своей системе.

Компиляция и связывание в Unix

Первоначально команда CC в Unix вызывала cfront. Однако cfront не удалось двигаться наравне с эволюцией C++, и его последний выпуск датируется 1993 г. В наши дни компьютеры на платформе Unix могут вообще не иметь компилятора, а могут иметь собственный компилятор или сторонний компилятор, будь-то коммерческий или свободно распространяемый вроде GNU g++. Во многих этих случаях (но не тогда, когда компилятора вообще нет) команда CC будет работать, вызывая компилятор, используемый в конкретной системе. Для простоты мы предполагаем, что команда CC доступна, но обратите внимание, что при последующем обсуждении она может быть заменена другой командой.

Команда CC используется для компиляции вашей программы. Имя команды вводится в верхнем регистре для того, чтобы отличить его от обычного компилятора C в Unix — cc. Компилятор CC является компилятором командной строки, что означает ввод команд компиляции в командной строке Unix.

Например, чтобы скомпилировать файл исходного кода C++ по имени spiffy.C, в строке приглашения Unix необходимо ввести следующую команду:

```
CC spiffy.C
```

Если, благодаря опыту или удаче в вашей программе не окажется ошибок, компилятор сгенерирует файл объектного кода с расширением o. В нашем случае компилятор создаст файл с именем

```
spiffy.o
```

Затем компилятор автоматически передает файл объектного кода системному редактору связей — программе, которая комбинирует ваш код с кодом библиотеки для построения исполняемого файла. По умолчанию исполняемому файлу назначается имя a.out. Если вы используете только один файл исходного кода, тогда редактор связей удалит файл spiffy.o, поскольку в нем больше нет необходимости. Чтобы запустить программу, достаточно ввести имя исполняемого файла:

```
a.out
```

Обратите внимание, что если вы компилируете новую программу, то новый исполняемый файл a.out заменит предыдущий файл a.out, который мог существовать. (Это происходит потому, что исполняемые файлы занимают достаточно много места, и замена старых исполняемых файлов позволяет сократить объем занимаемого дискового пространства.) Но если вы разрабатываете исполняемую программу, которую необходимо сохранить на будущее, воспользуйтесь Unix-командой mv, чтобы изменить имя исполняемого файла.

В языке C++, как и в C, одна программа может состоять из нескольких файлов. (Таковыми являются многие программы, рассматриваемые в главах 8–16 этой книги.) В этом случае компилировать программу можно, перечислив все эти файлы в командной строке:

```
CC my.C precious.C
```

При наличии нескольких файлов исходного кода компилятор не будет удалять файлы объектного кода. Так, если вы просто измените файл my.C, то повторную ком-

```
my.C precious.o
```

В результате файл `my.C` будет повторно скомпилирован и связан с ранее скомпилированным файлом `precious.o`.

Иногда возникает необходимость явным образом указывать некоторые библиотеки. Например, чтобы обратиться к функциям, определенным в библиотеке математических операций, может понадобиться добавить флаг `-lm` в командной строке:

```
CC usingmath.C -lm
```

Компиляция и связывание в Linux

В системах Linux чаще всего используется `g++` — компилятор GNU C++, разработанный Фондом свободного программного обеспечения (Free Software Foundation). Этот компилятор входит в состав большинства дистрибутивов Linux, однако может устанавливаться и отдельно. Компилятор `g++` работает подобно стандартному компилятору Unix. Например, в результате выполнения команды

```
g++ spiffy.cxx
```

будет создан исполняемый файл с именем `a.out`.

В некоторых версиях компилятора необходимо связываться с библиотекой C++:

```
g++ spiffy.cxx -lg++
```

Чтобы скомпилировать несколько файлов исходного кода, их достаточно перечислить в командной строке:

```
g++ my.cxx precious.cxx
```

В результате будет создан исполняемый файл по имени `a.out` и два файла объектного кода, `my.o` и `precious.o`. Если впоследствии вы измените только один файл исходного кода, например, `my.cxx`, то повторную компиляцию можно будет выполнить, используя `my.cxx` и `precious.o`:

```
g++ my.cxx precious.o
```

Компилятор GNU может работать на различных платформах, в том числе в режиме командной строки на ПК под управлением Windows и на различных платформах Unix-систем.

Компиляторы командной строки для режима командной строки Windows

Наименее затратный вариант компиляции программ на C++ в системах Windows заключается в том, чтобы загрузить свободно распространяемый компилятор командной строки, работающий в режиме командной строки Windows, при котором открывается MS-DOS-подобное окно. Бесплатно загружаемыми программами для Windows, которые включают компилятор GNU C++, являются Cygwin и MinGW; именем используемого в них компилятора является `g++`.

Для запуска компилятора `g++` сначала потребуется открыть окно командной строки. Программы Cygwin и MinGW делают это автоматически при запуске. Чтобы скомпилировать файл исходного кода по имени `great.cpp`, в командной строке введите следующую команду:

```
g++ great.cpp
```

В случае успешной компиляции будет сформирован исполняемый файл по имени `a.exe`.

Компиляторы для Windows

Компиляторов для Windows так много, при этом их модификации появляются настолько часто, что нет смысла рассматривать их все по отдельности. В настоящее время самым популярным является Microsoft Visual C++ 2010, который доступен также в виде бесплатной версии Microsoft Visual C++ 2010 Express. В Wikipedia (`http://en.wikipedia.org/wiki/List_of_compilers`) представлен исчерпывающий список компиляторов для множества платформ, включая Windows. Несмотря на разные проектные решения и цели, большинство компиляторов C++ для Windows обладают общими характеристиками.

Как правило, для программы требуется создать проект и добавить в него один или несколько файлов, составляющих программу. Каждый производитель предлагает IDE-среду с набором меню и, возможно, с программой автоматизированного помощника, которую удобно использовать в процессе создания проекта. Очень важно определиться с тем, к какому типу будет относиться создаваемая вами программа. Обычно компилятор предлагает несколько вариантов, среди которых приложение для Windows, приложение Windows MFC, динамически подключаемая библиотека, элемент управления ActiveX, программа, выполняемая в режиме DOS или в символьном режиме, статическая библиотека или консольное приложение. Некоторые из них могут быть доступны в форме 64- и 32-разрядных версий.

Поскольку программы в этой книге являются обобщенными, вы должны избегать вариантов, требующих кода для определенной платформы, например, приложение для Windows. Вместо этого нужно запускать программу в символьном режиме. Выбор режима зависит от компилятора. В общем случае необходимо искать такие варианты, как консольное, символьное или DOS-приложение, и пробовать их. Например, в среде Microsoft Visual C++ 2010 выберите опцию Win32 Console Application (Консольное приложение Win32), щелкните на Application Settings (Настройки приложения) и выберите вариант Empty Project (Пустой проект). В C++Builder XE выберите вариант Console Application (Консольное приложение) в разделе C++Builder Projects (Проекты C++Builder).

После того как проект настроен, вы должны скомпилировать и скомпоновать свою программу. В IDE-среде обычно предлагается несколько вариантов, такие как Compile (Компилировать), Build (Построить), Make (Построить), Build All (Построить все), Link (Скомпоновать), Execute (Выполнить), Run (Выполнить) и Debug (Отладить) (но не обязательно все варианты сразу).

- Compile обычно означает компиляцию кода в открытом в настоящий момент файле.
- Build или Make обычно означает компиляцию кода для всех файлов исходного кода, входящих в состав данного проекта. Часто этот процесс является инкрементным. То есть, если в проекте было три файла, и вы изменили только один из них, то повторно скомпилирован будет только этот файл.
- Build All обычно означает компиляцию всех файлов исходного кода с самого начала.
- Как уже было сказано ранее, Link означает объединение скомпилированного исходного кода с необходимым библиотечным кодом.
- Run или Execute означает запуск программы. Обычно если вы еще не завершили выполнение предыдущих этапов, команда Run выполнит их перед запуском программы.

- Debug означает запуск программы с возможностью ее пошагового выполнения.
- Компилятор может поддерживать создание версий Debug (Отладочная) и Release (Выпуск). Версия Release содержит дополнительный код, который увеличивает размер программы и замедляет ее выполнение, но зато делает доступными средства отладки.

Если вы нарушите какое-то правило языка, компилятор выдаст сообщение об ошибке и укажет на строку кода, в которой она была найдена. К сожалению, если вы еще недостаточно хорошо знаете язык программирования, то понять смысл этого сообщения порой бывает трудно. В некоторых случаях действительная ошибка может находиться перед указанной строкой, а бывает так, что единственная ошибка порождает цепочку сообщений об ошибках.

Совет

Исправление ошибок начинайте с самой первой. Если вы не можете ее найти в указанной строке, проверьте предыдущую строку кода.

Имейте в виду, что факт принятия программы определенным компилятором вовсе не означает, что эта программа является допустимой программой на C++. И то, что определенный компилятор отклоняет программу, не обязательно означает, что эта программа не является допустимой программой C++. Однако современная компиляторы в большей степени соответствуют принятому стандарту, нежели их предшественники несколько лет тому назад. Кроме того, компиляторы, как правило, имеют опции для управления строгостью компиляции.

Совет

Время от времени компиляторы, не завершив процесс создания программы, генерируют бессмысленные сообщения об ошибках, которые невозможно устранить. В подобных ситуациях следует воспользоваться командой Build All, чтобы начать процесс компиляции с самого начала. К сожалению, отличить эту ситуацию от другой, более распространенной, когда сообщение об ошибке только кажется бессмысленным, достаточно трудно.

Обычно IDE-среда позволяет запускать программу во вспомогательном окне. В некоторых IDE-средах это окно закрывается после завершения выполнения программы, а в некоторых оно остается открытым. Если ваш компилятор закрывает окно, вы не успеете увидеть вывод программы, если только не умеете очень быстро читать и не обладаете фотографической памятью. Чтобы увидеть результат выполнения, в конце программы необходимо ввести следующий код:

```
    cin.get(); // добавьте этот оператор
    cin.get(); // и, возможно, этот тоже
    return 0;
}
```

Оператор `cin.get()` читает следующее нажатие клавиши, поэтому программа будет находиться в режиме ожидания до тех пор, пока вы не нажмете клавишу <Enter>. (Вплоть до нажатия <Enter> никакие нажатия клавиш программе не отправляются, поэтому нажимать другие клавиши не имеет смысла.) Второй оператор необходим на случай, если программа оставит необработанным нажатие клавиши после предыдущего ввода информации в программе. Например, при вводе числа вы нажимаете соответствующую клавишу, а затем <Enter>. Программа может прочитать число, но

оставить необработанным нажатие клавиши <Enter>, и затем читает его в первом операторе `cin.get()`.

C++ в Macintosh

Компания Apple в настоящее время поставляет платформу разработки под названием Xcode вместе с операционной системой Mac OS X. Эта платформа является бесплатной, но обычно предварительно не устанавливается. Ее можно установить с дистрибутивных дисков ОС или загрузить за номинальную плату из Apple. (Имейте в виду, что объем загрузки превышает 4 Гбайт.) Платформа Xcode не только предоставляет IDE-среду, поддерживающую множество языков программирования, она также устанавливает пару компиляторов — `g++` и `clang`, — которые могут использоваться как программы командной строки в режиме Unix, доступном через утилиту Terminal.

> **Совет**
>
> Для IDE-сред: чтобы сэкономить время, можно использовать только один проект для всех примеров программ. Просто удалите файл с предыдущим примером исходного кода из списка проекта и затем добавьте новый исходный код. Это сохраняет время, усилия и дисковое пространство.

Резюме

Компьютеры стали более мощными, а компьютерные программы — большими и сложными. Как результат, компьютерные языки стали более развитыми и теперь они упрощают управление процессом написания программ. Язык C наделен такими средствами, как управляющие структуры и функции, с помощью которых можно расширить возможности управления ходом выполнения программы и реализовать более структурированный, модульный подход к написанию программ. Для данных инструментов в C++ поддерживается объектно-ориентированное программирование (ООП), а также обобщенное программирование. Благодаря этому открываются возможности для еще более высокой степени модульности и повторного использования кода, что позволяет сократить время на разработку и повысить надежность создаваемых программ.

Популярность языка программирования C++ привела к появлению большого количества реализаций для множества компьютерных платформ; стандарты C++ ISO (C++98/03 и C++11) обеспечивают основу для взаимной совместимости этих реализаций. В стандартах указано, какими возможностями должен обладать язык, как он себя должен вести, какими должны быть библиотеки функций, классы и шаблоны. Стандарты поддерживают идею переносимого языка, программы на котором могут работать на множестве различных вычислительных платформ и в различных реализациях языка.

Чтобы создать программу на языке C++, вы создаете один или несколько файлов исходного кода, написанного на C++. Это текстовые файлы, которые необходимо компилировать и компоновать для формирования файлов на машинном языке, содержащих исполняемые программы. Эти задачи часто выполняются в IDE-среде, которая предлагает текстовой редактор для подготовки файлов исходного кода, компилятор и компоновщик для построения исполняемых файлов, а также другие ресурсы наподобие средств управления проектом и отладкой. Однако те же самые задачи могут быть решены также и в командной строке, в которой соответствующие инструменты вызываются по отдельности.

2

Приступаем к изучению C++

В ЭТОЙ ГЛАВЕ...

- Создание программы на C++
- Общий формат программы на C++
- Директива `#include`
- Функция `main()`
- Использование объекта `cout` для вывода
- Помещение комментариев в программу на C++
- Использование манипулятора `endl`
- Объявление и использование переменных
- Использование объекта `cin` для ввода
- Определение и использование простых функций

Приступая к строительству простого дома, вы начинаете с его фундамента и каркаса. Если с самого начала работы вы не сформируете цельную структуру, то впоследствии не сможете вставить окна или дверные коробки, нельзя будет соорудить обзорные купола или выстроить танцевальный зал с паркетным покрытием. Это же относится и к изучению компьютерного языка программирования: прежде всего, необходимо освоить базовую структуру программы, и только после этого можно переходить к изучению деталей, например, циклов и объектов. В этой главе рассматривается базовая структура программы на C++ и ряд дополнительных вопросов. Часть материала будет посвящена функциям и классам, которым в последующих главах будет уделено особое внимание. (Идея заключается в том, чтобы постепенно, по ходу изложения материала, вводить основные положения, а затем рассматривать каждый вопрос детально.)

Первые шаги в C++

Давайте попробуем написать простую программу на языке C++, которая будет выводить текст некоторого сообщения на экран монитора. В листинге 2.1 для символьного вывода применяется объект `cout`. Исходный код содержит строки комментариев для читателя, которые отмечаются парой символов `//`; эти символы компилятор игнорирует. Язык программирования C++ *чувствителен к регистру символов*; это означает, что символы в верхнем и нижнем регистре считаются разными. Поэтому должен использоваться в точности тот же регистр, что применяется в примерах. Например, в приведенной далее программе используется `cout`, поэтому если вы введете `Cout` или `COUT`, компилятор отклонит это и сообщит о наличии неизвестных идентификаторов. (Компилятор чувствителен также и к орфографии, поэтому не пытайтесь указывать `kout`, `coot` и т.п.) Расширение `cpp` имени файла чаще всего используется для обозначения программы на C++; встречаются также и другие расширения, о чем говорилось в главе 1.

Листинг 2.1. `myfirst.cpp`

```
// myfirst.cpp -- выводит сообщение на экран

#include <iostream>                          // директива препроцессора
int main()                                   // заголовок функции
{                                            // начало тела функции
   using namespace std;                      // делает видимыми определения
   cout << "Come up and C++ me some time."; // сообщение
   cout << endl;                             // начало новой строки
   cout << "You won't regret it!" << endl;  // дополнительный вывод
   return 0;                                 // завершение функции main()
                                             // конец тела функции
```

Подгонка кода программы

Может оказаться, что для запуска примеров в вашей системе код должен быть изменен. Чаще всего это связано с конкретной средой программирования. Некоторые оконные среды запускают программу в отдельном окне и затем автоматически закрывают его при завершении выполнения программы. Как уже упоминалось в главе 1, можно сделать так, чтобы окно оставалось открытым до тех пор, пока пользователь не нажмет клавишу; для этого перед оператором `return` необходимо добавить следующую строку кода:

```
cin.get();
```

В некоторых программах необходимо добавлять две строки, чтобы оставить окно открытым до нажатия клавиши. Оператор `cin.get()` более подробно рассматривается в главе 4.

В очень старых системах могут не поддерживаться средства, введенные стандартом C++98.

Некоторые программы требуют наличия компилятора с определенным уровнем поддержки стандарта C++11. Этот факт будет специальным образом отмечен и по возможности будет предоставлен альтернативный код, отличный от C++11.

После выбора подходящего редактора для копирования этой программы (либо использования файлов исходного кода, доступных для загрузки на веб-сайте издательства), с помощью компилятора C++ можно создать исполняемый код, как объяснялось в главе 1. Ниже показан результат выполнения скомпилированной программы из листинга 2.1:

```
Come up and C++ me some time.
You won't regret it!
```

Ввод и вывод в языке C

Если у вас есть опыт программирования на языке C, то использование объекта `cout` вместо функции `printf()` может показаться странным. На самом деле, в C++ можно применять как `printf()`, так и `scan()` и другие стандартные функции ввода-вывода языка C, которые становятся доступными благодаря включению обычного C-файла `stdio.h`. Однако эта книга посвящена C++, поэтому мы применяем средства ввода-вывода C++, которые являются более полезными, нежели средства C.

Программы на C++ конструируются из строительных блоков, называемых *функциями*. Обычно программа систематизируется в набор главных задач, для обработки которых проектируются отдельные функции. Пример, представленный в листинге 2.1, настолько прост, что состоит всего лишь из одной функции `main()`.

Пример `myfirst.cpp` содержит следующие элементы:

- комментарии, обозначаемые с помощью `//`;
- директива препроцессора `#include`;
- заголовок функции: `int main()`;
- директива `using namespace`;
- тело функции, ограниченное фигурными скобками `{` и `}`;
- операторы, которые используют объект C++ `cout` для отображения сообщения;
- оператор возврата для прекращения выполнения функции `main()`.

Давайте обсудим все эти элементы более подробно. Лучше всего начать с функции `main()`, поскольку некоторые предшествующие ей элементы, такие как директива препроцессора, будет проще понять после рассмотрения того, что делает функция `main()`.

Возможности функции `main()`

Пример программы из листинга 2.1 имеет следующую фундаментальную структуру:

```
int main()
{
   операторы
   return 0;
}
```

В этих строках говорится о том, что существует функция по имени `main()`, и они описывают ее поведение. Вместе эти строки образуют *определение функции*. Это определение состоит из двух частей: первой строки, `int main()`, которая называется *заголовком функции*, и частью, заключенной в скобки ({ и }), которая называется *телом функции*. (В стандарте ISO эти скобки называются фигурными.) На рис. 2.1 приведена графическое представление функции `main()`. В заголовке функции вкратце описан ее интерфейс с остальной частью программы, а в теле функции содержатся компьютерные инструкции о том, что функция должна делать. В языке C++ каждая полная инструкция называется *оператором*. Каждый оператор должен завершаться точкой с запятой, поэтому не забывайте ставить ее при наборе примеров.

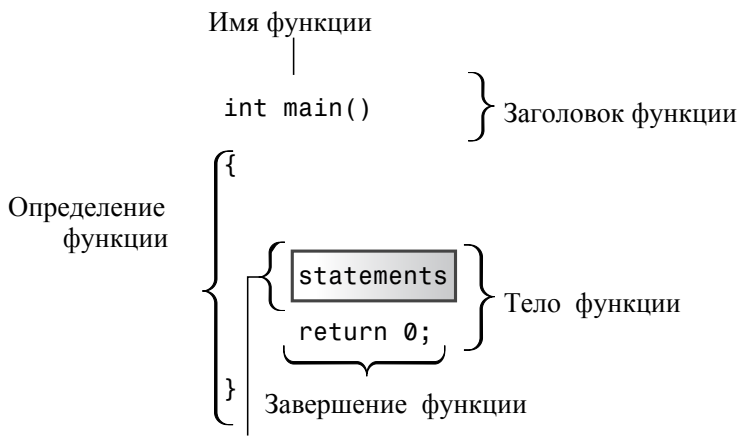


Рис. 2.1. *Функция* `main()`

Финальный оператор функции `main()` называется *оператором возврата*. Его назначение — завершить выполнение функции. В этой главе мы еще вернемся к нему.

Операторы и точки с запятыми

Оператор представляет действие, которое должно быть выполнено. Чтобы понять ваш исходный код, компилятор должен знать, где заканчивается один оператор и начинается другой. В некоторых языках программирования используется разделитель между операторами. В языке FORTRAN, например, для разделения операторов друг от друга применяется символ конца строки. В языке Pascal для разделения операторов используется точка с запятой. В языке Pascal точку с запятой можно в некоторых случаях опускать, например, после оператора и перед `END`, когда на самом деле разделение двух операторов не происходит. (Прагматики и минималисты могут не согласиться с тем, что *можно* — это значит *нужно*.) Однако в языке C++, в отличие от C, точка с запятой больше используется в качестве *терминатора*, или признака завершения, а не как разделительный знак. Разница заключается в том, что точка с запятой, действующая как терминатор, является *частью* оператора, а не маркером *между* операторами. На практике в языке C++ точку с запятой никогда нельзя опускать.

Заголовок функции как интерфейс

Сейчас самое главное усвоить следующее правило: синтаксис языка программирования C++ требует начинать определение функции `main()` с заголовка `int main()`. Синтаксис заголовка функции детально рассматривается позже, в разделе "Функции", а пока что для тех, кому просто не терпится удовлетворить свой интерес, мы вкратце расскажем о сути этого заголовка.

Функция в языке C++ активизируется, или *вызывается*, другой функцией, и заголовок функции описывает интерфейс между функцией и вызывающей ее функцией. Та часть, которая предшествует имени функции, называется *возвращаемым типом функции*; в ней описывается информационный поток, поступающий из функции обратно к функции, которая ее вызвала. Та часть, которая заключена в скобки после имени функции, называется *списком аргументов* или *списком параметров*; в ней описывается информационный поток, который поступает из вызывающей функции к вызываемой функции.Применительно к `main()` это описание будет немного отличаться, поскольку она, как правило, не вызывается из других частей программы.

Обычно к функции `main()` обращается код запуска, добавляемый компилятором в вашу программу – он нужен в качестве связующего звена между программой и ОС (Unix, Windows 7, Linux и т.п.). По сути, заголовок функции описывает интерфейс между функцией `main()` и ОС.

Давайте рассмотрим описание интерфейса `main()`, начиная с части `int`. Функция C++, которую вызывает другая функция, возвращает значение в активизирующую (вызывающую) функцию. Это значение называется *возвращаемым значением*. В данном случае функция `main()` может возвращать целочисленное значение, на что указывает ключевое слово `int`. Далее обратите внимание на пустые скобки. В общем случае функция C++ может передавать информацию функции, которую вызывает. Эту информацию описывает часть заголовка функции, заключенная в скобки. В нашем случае пустые скобки означают, что `main()` не принимает никакой информации, или, если обратиться к общепринятой терминологии, функция `main()` не принимает аргументов. (Утверждение о том, что функция `main()` не принимает аргументов, не означает, что она не имеет смысла. Под *аргументом* программисты подразумевают информацию, которая передается из одной функции в другую.)

Короче говоря, заголовок

```
int main()
```

говорит о том, что функция `main()` возвращает целочисленное значение в вызывающую ее функцию, и что функция `main()` не принимает никакой информации от вызывающей ее функции.

Во многих существующих программах используется классическая форма записи заголовка функции в стиле C:

```
main()          // исходный стиль C
```

В классическом C опускание возвращаемого типа равнозначно тому, что функция имеет тип `int`. Однако в языке C++ от такого подхода отказались.

Можно использовать и такой вариант:

```
int main(void)  // самый подробный стиль
```

Использование ключевого слова `void` в скобках – это явный способ указать, что данная функция не принимает аргументов. В C++ (но не в C) принято, что пустые скобки равнозначны использованию ключевого слова `void`. (В языке C пустые скобки означают, что вы ничего не сообщаете о наличии аргументов.)

Некоторые программисты используют следующий заголовок и опускают оператор возврата:

```
void main()
```

Это логически имеет смысл, поскольку возвращаемый тип `void` означает, что функция не возвращает значения. Однако хотя данный вариант работает в некоторых системах, он не является частью стандарта C++. Таким образом, в других систе-

мах он даст сбой. Лучше всего не применять эту форму записи, а использовать форму, соответствующую стандарту C++; особых усилий это не потребует.

И, наконец, стандарт ISO C++ идет на уступки тем программистам, которым не особо нравится необходимость ставить оператор возврата в конце функции `main()`. Если компилятор достиг завершения функции `main()`, не встретив при этом оператора возврата, результат будет таким же, как если бы `main()` заканчивалась следующим оператором:

```
return 0;
```

Неявный возврат возможен только для функции `main()`, но не для любой другой функции.

Почему функции `main()` нельзя назначать другое имя

Существует убедительная причина назначить функции в программе `myfirst.cpp` имя `main()`: вы обязаны поступать так. Обычно любая программа C++ требует наличия функции по имени `main()`. (Не `Main()`, `MAIN()` или, скажем, `mane()`. Не забывайте о чувствительности к регистру и орфографии.) Поскольку в программе `myfirst.cpp` имеется только одна функция, то она и должна взять на себя ответственность и стать `main()`. Выполнение программы на C++ всегда начинается с функции `main()`. Таким образом, если в программе функция `main()` отсутствует, то такая программа рассматривается как неполная и компилятор выдаст сообщение об отсутствии определения функции `main()`.

Существует ряд исключений. Например, при программировании для Windows вы можете написать модуль динамически подключаемой библиотеки (Dynamic Link Library — DLL). Эта библиотека содержит код, который могут использовать другие Windows-программы. Поскольку модуль DLL не является автономной программой, для него функция `main()` не нужна. Программы, предназначенные для работы в специальных средах, таких как контроллер в автоматическом устройстве, могут не нуждаться в функции `main()`. Некоторые среды для программирования предоставляют скелетную программу, вызывающую нестандартную функцию наподобие `_tmain()`; в этом случае имеется скрытая функция `main()`, которая вызывает `_tmain()`. Однако обычная автономная программа должна иметь `main()`; в этой книге рассматриваются именно такие программы.

Комментарии в языке C++

В C++ комментарий обозначается двумя косыми чертами (`//`). *Комментарий* — это примечание, написанное программистом для пользователя программы, которое обычно идентифицирует ее раздел или содержит пояснения к определенному коду. Компилятор игнорирует комментарии. Хотя он знает C++ не хуже вас, понимать комментарии он не умеет. Поэтому для него листинг 2.1 будет выглядеть следующим образом:

```
#include <iostream>
int main()
{
   using namespace std;
   cout << "Come up and C++ me some time.";
   cout << endl;
   cout << "You won't regret it!" << endl;
   return 0;
}
```

Комментарий в C++ начинается с символов // и простирается до конца строки. Комментарий может занимать одну строку целиком, а может находиться в строке вместе с кодом. Обратите внимание на первую строку в листинге 2.1:

```
// myfirst.cpp -- выводит сообщение на экран
```

В этой книге все программы начинаются с комментария, в котором указывается имя файла исходного кода и краткое описание программы. Как упоминалось в главе 1, расширение имени файла исходного кода зависит от реализации C++. В других реализациях могут использоваться имена вроде `myfirst.C` или `myfirst.cxx`.

Совет

Используйте комментарии для документирования своих программ. Чем сложнее программа, тем более ценными будут ваши комментарии. Они помогут не только другим пользователям разобраться с вашим кодом, но и вы сами сможете вспомнить, что он делает, по прошествии некоторого времени.

Комментарии в стиле C

Язык C++ распознает комментарии в стиле C, заключенные между символами /* и */:

```
#include <iostream>  /* комментарий в стиле C */
```

Поскольку комментарий в C завершается символами */, а не в конце строки, он может занимать несколько строк. В программах допускается использование обоих стилей комментариев. Однако мы рекомендуем придерживаться стиля C++. Поскольку он не требует отслеживания порядка чередования конечного и начального символов, вероятность допустить ошибку при вводе будет невелика. В стандарте C99 для языка C добавлен комментарий //.

Препроцессор C++ и файл iostream

Для начала давайте кратко рассмотрим то, что вы должны знать обязательно. Если ваша программа предназначена для использования обычных средств ввода и вывода в C++, вы предоставляете следующие две строки:

```
#include <iostream>
using namespace std;
```

Для второй строки существуют альтернативные варианты, но в данный момент мы стремимся все максимально упростить. (Если используемый вами компилятор отклоняет эти строки кода, значит, он не совместим со стандартом C++98; в таком случае с примерами, приведенными в этой книге, будут возникать и другие проблемы.) Это все, что вам нужно знать для обеспечения работоспособности программы. А теперь давайте рассмотрим все более подробно.

В языке C++, как и в C, используется *препроцессор*. Препроцессор — это программа, которая выполняет обработку файла исходного кода перед началом собственно компиляции. (В некоторых реализациях языка C++, как упоминалось в главе 1, для преобразования программы C++ в C используется программа транслятора. Хотя этот транслятор можно считать определенной разновидностью препроцессора, мы его не рассматриваем; мы говорим о препроцессоре, обрабатывающем директивы, имена которых начинаются с символа #.) Для вызова препроцессора ничего особенного делать не нужно. Он запускается автоматически во время компиляции программы.

В листинге 2.1 использовалась директива `#include`:

```
#include <iostream> // директива препроцессора
```

Эта директива заставляет препроцессор добавить содержимое файла iostream в вашу программу. Добавление или замена текста в исходном коде перед его компиляцией является одним из обычных действий препроцессора.

Может возникнуть вопрос: зачем добавлять содержимое файла iostream в программу? Причина состоит в необходимости коммуникаций между программой и внешним миром. Часть io в iostream означает *input* (ввод) — входные данные программы, и *output* (вывод) — информация, передаваемая из программы. Схема ввода-вывода в C++ включает несколько определений, которые можно найти в файле iostream. В вашей первой программе эти определения необходимы для использования объекта cout при выводе сообщения на экран. В соответствии с директивой #include, содержимое файла iostream будет отправлено компилятору вместе с содержимым вашего файла. В действительности содержимое файла iostream заменяет строку #include <iostream> в программе. Ваш исходный файл не изменяется, а результирующий файл, сформированный из вашего файла и iostream, передается на следующий этап компиляции.

На заметку!

Программы, в которых для ввода и вывода используются объекты cin и cout, должны включать файл iostream.

Имена заголовочных файлов

Файлы, подобные iostream, называются *включаемыми файлами* (поскольку они включаются в другие файлы) или *заголовочными файлами* (т.к. они включаются в самом начале файла). Компиляторы C++ поставляются вместе с множеством заголовочных файлов, каждый из которых поддерживает отдельное семейство возможностей. В языке C для заголовочных файлов использовалось расширение h, благодаря которому можно было просто распознать тип файла по его имени. Например, заголовочный файл math.h в C поддерживает разнообразные математические функции языка C. Первоначально так было и в языке C++. Например, заголовочный файл, поддерживающий ввод и вывод, имел имя iostream.h. Однако использование C++ претерпело некоторые изменения. Сейчас расширение h зарезервировано для старых заголовочных файлов C (которые по-прежнему могут применяться в программах C++), тогда как заголовочные файлы в C++ не имеют расширений. Существуют также заголовочные файлы C, которые можно преобразовать в заголовочные файлы C++. Имена этих файлов были изменены: было исключено расширение h (для формирования стиля C++) и добавлен префикс c (отражающий, что файл относится к языку C). Например, версией math.h в C++ является заголовочный файл cmath. Иногда версии заголовочных файлов C в языках C и C++ одинаковы, а в других случаях новая версия может несколько отличаться. Для чистых заголовочных файлов C++, таких как iostream, отсутствие h является не просто косметическим изменением, но свидетельством использования пространства имен, о которых речь пойдет в следующем разделе.

В табл. 2.1 приведены соглашения по именованию для заголовочных файлов.

Учитывая традицию применения в C различных расширений имен файлов для отражения разных типов файлов, представляется целесообразным использовать для обозначения заголовочных файлов в C++ некоторое специальное расширение, например, .hx или .hxx. Так считают и в комитете ANSI/ISO. Однако договориться об этом оказалось очень сложно, поэтому, в конечном счете, все осталось по-прежнему.

Таблица 2.1. Соглашения по именованию заголовочных файлов

Тип заголовка	Соглашение	Пример	Комментарии
Старый стиль C++	Заканчивается на `.h`	`iostream.h`	Используется в программах на C++
Старый стиль C	Заканчивается на `.h`	`math.h`	Используется в программах на C и C++
Новый стиль C++	Без расширения	`iostream`	Используется в программах на C++, использует пространство имен `std`
Преобразованный C	Префикс `c`, без расширения	`cmath`	Используется в программах на C++, может использовать средства, не характерные для C, такие как пространство имен `std`

Пространства имен

Если вместо `iostream.h` вы применяете `iostream`, тогда необходимо использовать следующую директиву пространства имен, чтобы определения в `iostream` были доступны в программе:

```
using namespace std;
```

Это называется *директивой* `using`. Сейчас самое главное — просто запомнить ее. Мы еще вернемся к этому вопросу (в главе 9, например), а пока давайте кратко рассмотрим, что собой представляют пространства имен.

Поддержка пространства имен — это средство C++, предназначенное для упрощения разработки крупных программ и программ, в которых комбинируется существующий код от нескольких поставщиков, а также для помощи в организации таких программ. Одна из потенциальных проблем заключается в том, что вы можете работать с двумя готовыми продуктами, в каждом из которых присутствует, скажем, функция `wanda()`. При использовании функции `wanda()` компилятор не будет знать, какая конкретно версия этой функции имеется в виду. Средство пространств имен позволяет поставщику упаковать свой продукт в модуль, называемый *пространством имен*. Вы, в свою очередь, можете использовать название этого пространства имен для указания, продукт какого производителя вам необходим. Так, например, компания Microflop Industries может поместить свои определения в пространство имен `Microflop`. Тогда полное имя функции `wanda()` будет выглядеть как `Microflop::wanda()`. Аналогично, `Piscine::wanda()` может обозначать версию функции `wanda()` от компании Piscine Corporation. Таким образом, для различения версий функции в программе вы можете использовать пространства имен:

```
Microflop::wanda("go dancing?");        // использует версию из
                                        // пространства имен Microflop
Piscine::wanda("a fish named Desire");  // использует версию из
                                        // пространства имен Piscine
```

Соблюдая такой стиль, классы, функции и переменные, являющиеся стандартными компонентами компиляторов C++, теперь находятся в пространстве имен `std`. Это относится к заголовочным файлам без расширения `h`. Это означает, например, что переменная `cout`, используемая для вывода и определенная в `iostream`, на самом деле называется `std::cout`, а `endl` в действительности выглядит как `std::endl`.

Таким образом, директиву using можно опустить и записать код следующим образом:

```
std::cout << "Come up and C++ me some time.";
std::cout << std::endl;
```

Однако многие программисты иногда отказываются преобразовывать код, написанный до появления пространств имен, в котором используются iostream.h и cout, в код с пространствами имен, если это требует значительных усилий. Вот здесь и приходит на помощь директива using. Следующая строка означает, что вы можете применять имена, определенные в пространстве имен std, без префикса std::

```
using namespace std;
```

Приведенная директива using делает доступными все имена в пространстве имен std. В настоящее время это рассматривается как проявление лени, которое приводит к возникновению проблем в крупных проектах. Более удачные подходы предполагают использование квалификатора std:: или объявления using для обеспечения доступа только к отдельным именам:

```
using std::cout;      // делает доступным cout
using std::endl;      // делает доступным endl
using std::cin;       // делает доступным cin
```

Если эти директивы применить вместо приведенной ниже строки, то cin и cout можно использовать без std::

```
using namespace std; // ленивый подход; доступны все имена
```

Однако если потребуются другие имена из iostream, их придется добавлять в список using по отдельности. В этой книге используется ленивый подход, и это объясняется двумя причинами. Во-первых, для простых программ нет особой разницы, какой применяется прием управления пространствами имен. Во-вторых, целью книги является акцентирование внимания на более базовых аспектах языка C++. Далее в этой книге вы встретите и другие приемы управления пространствами имен.

Вывод в C++ с помощью cout

Давайте посмотрим, как вывести сообщение на экран. В программе myfirst.cpp используется следующий оператор C++:

```
cout << "Come up and C++ me some time.";
```

Часть, заключенная в двойные кавычки – это сообщение, которое необходимо вывести на экран. В C++ любая последовательность символов, заключенных в двойные кавычки, называется *символьной строкой*, по-видимому, из-за того, что она состоит из множества символов, собранных в одну большую конструкцию. Запись << означает, что оператор отправляет строку в cout; символы указывают на направление передачи информации. А что такое cout? Это предопределенный объект, который знает о том, как отображать разнообразные элементы, включая строки, цифры и индивидуальные символы. (Как вы помните из главы 1, *объект* представляет собой экземпляр класса, а *класс* определяет способ хранения и использования данных.)

Возможно, приступать к использованию объектов еще рано, потому что они еще подробно не рассматривались. На самом деле, в приведенном примере продемонстрирована одна из сильных сторон объектов. Для работы с объектом вы не обязаны знать его внутреннюю структуру. Все, что вам необходимо знать – это его интерфейс, т.е. способ его использования. Объект cout имеет простой интерфейс. Если string представляет строку, то для ее вывода на экран можно сделать следующее:

```
cout << string;
```

Вот и все, что требуется для отображения строки на экране. А сейчас давайте посмотрим, как этот процесс можно описать с точки зрения концептуального представления C++. В этом представлении вывод рассматривается как поток, т.е. последовательность символов, передаваемых из программы. Этот поток представляет объект `cout`, свойства которого определены в файле `iostream`. Свойства объекта `cout` включают операцию вставки (<<), которая добавляет в поток данные, указанные в правой части. Рассмотрим следующий оператор (обратите внимание на завершающую точку с запятой):

```
cout << "Come up and C++ me some time.";
```

В выходной поток будет помещена строка `"Come up and C++ me some time."`. Таким образом, вы можете сказать, что ваша программа не выводит на экран сообщение, а вставляет строку в поток вывода. В чем-то это звучит более выразительно (рис. 2.2).

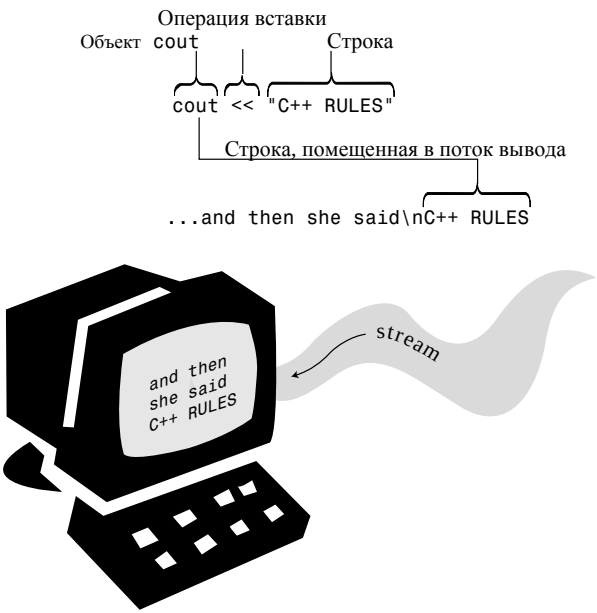


Рис. 2.2. Использование `cout` для отображения строки

Несколько слов о перегрузке операций

Если вы начали изучать язык C++, имея опыт программирования на C, то, скорее всего, заметили, что операция вставки (<<) выглядит в точности так же, как побитовая операция сдвига влево (<<). Это пример *перегрузки операций*, когда один и тот же символ операции может трактоваться по-разному. Чтобы выяснить назначение выполняемой операции, компилятор обращается к контексту. В языке C тоже есть примеры перегрузки операций. Например, символ `&` представляет как адресную операцию, так и побитовую операцию `AND` ("И"). Символом `*` обозначается умножение и разыменование указателя. Важный момент связан не с конкретными функциями этих операций, а с тем, что один и тот же символ трактуется по-разному, и компилятор определяет соответствующее назначение на основе контекста. (В повседневной жизни мы тоже нередко обращаемся к контексту, чтобы правильно понять сказанную фразу.) В языке C++ концепция перегрузки операций расширена, благодаря чему можно переопределять предназначение операций для классов — т.е. типов, определяемых пользователем.

Манипулятор endl

Теперь давайте рассмотрим еще одну строку из листинга 2.1:

```
cout << endl;
```

Здесь endl — это специальное обозначение в C++, которое представляет важное понятие начала новой строки. Вставка endl в поток вывода означает, что курсор на экране будет перемещен на начало следующей строки. Специальные обозначения наподобие endl, которые имеют определенное значение для cout, называются *манипуляторами*. Как и cout, манипулятор endl определен в заголовочном файле iostream и является частью пространства имен std.

Обратите внимание, что при выводе строки cout не переходит автоматически на следующую строку, поэтому первый оператор cout в листинге 2.1 оставляет курсор в позиции после точки в конце строки вывода. Вывод для каждого оператора cout начинается с той позиции, где был закончен последний вывод, поэтому если опустить манипулятор endl, результат будет таким:

```
Come up and C++ me some time.You won't regret it!
```

Обратите внимание, что Y следует сразу за точкой. Давайте рассмотрим еще один пример. Предположим, что вы ввели следующий код:

```
cout << "The Good, the";
cout << "Bad, ";
cout << "and the Ukulele";
cout << endl;
```

В результате его выполнения будет выведена следующая строка:

```
The Good, theBad, and the Ukulele
```

Здесь легко заметить, что одна из строк начинается сразу после окончания предыдущей строки. Чтобы поместить между двумя строками пробел, необходимо включить его в одну из строк. (Не забывайте, что эти примеры должны быть помещены в завершенную программу, с заголовком функции main() и открывающей и закрывающей фигурными скобками, иначе выполнить их не удастся.)

Символ новой строки

Обозначить новую строку в выводе в C++ можно и старым способом — посредством обозначения \n, принятого в языке C:

```
cout << "What's next?\n"; // \n обозначает начало новой строки
```

Комбинация \n рассматривается как один символ, называемый символом *новой строки*. При отображении строки использование \n сокращает количество печатаемых символов, чем манипулятор endl:

```
cout << "Jupiter is a large planet.\n";       // отображает текст,
                                               // переходит на следующую строку
cout << "Jupiter is a large planet." << endl; // отображает текст,
                                               // переходит на следующую строку
```

С другой стороны, если вы хотите сгенерировать отдельную новую строку, то в каждом из случаев придется набирать одинаковое количество знаков; большинство программистов считают, что набирать endl гораздо удобнее:

```
cout << "\n";    // начинает новую строку
cout << endl;    // начинает новую строку
```

В этой книге используется встроенный символ новой строки (\n) при отображении строк в кавычках, а во всех других случаях – манипулятор endl. Одно из отличий состоит в том, что endl гарантирует сброс вывода (в данном случае немедленное отображение на экране) перед продолжением выполнения программы. В случае использования "\n" такой гарантии не будет, а это значит, что в некоторых системах при определенных условиях возможно отсутствие отображения приглашения до тех пор, пока не будет введена запрашиваемая информация.

Символ новой строки является примером специальных комбинаций клавиш, называемых "управляющими последовательностями"; речь о них пойдет в главе 3.

Форматирование исходного кода C++

Код в некоторых языках программирования, таких как FORTRAN, является построчным, т.е. в одной строке находится один оператор. В этих языках для разделения операторов служит возврат каретки (генерируемый нажатием клавиши <Enter> или <Return>). В языке C++ для обозначения завершения каждого оператора служит точка с запятой. Поэтому в C++ возврат каретки можно трактовать точно так же, как пробел или табуляцию. Это означает, что в C++ можно использовать пробел там, где можно было бы использовать возврат каретки, и наоборот. Поэтому вы могли бы записывать один оператор в нескольких строках или ставить несколько операторов в одной строке. Например, код myfirst.cpp можно было бы переформатировать следующим образом:

```
#include <iostream>
      int
main
() {   using
   namespace
            std; cout
                 <<
"Come up and C++ me some time."
;  cout <<
endl; cout <<
"You won't regret it!" <<
endl;return 0; }
```

Этот код выглядит неуклюже, но является допустимым. При вводе кода необходимо соблюдать некоторые правила. В частности, в C и C++ нельзя размещать пробел, табуляцию или возврат каретки в середине элемента, например в имени, и не допускается помещать возврат каретки в середину строки. Далее показаны примеры неправильного ввода:

```
int ma in()                    // НЕПРАВИЛЬНО – пробел в имени
re
turn 0;                        // НЕПРАВИЛЬНО – возврат каретки в слове
cout << "Behold the Beans
of Beauty!";                   // НЕПРАВИЛЬНО – возврат каретки в строке
```

Лексемы и пробельные символы в исходном коде

Лексемами называются неделимые элементы в строке кода (рис. 2.3). Как правило, для разделения лексем друг от друга используется пробел, табуляция или возврат каретки, которые все вместе называются *пробельными символами*. Некоторые одиночные символы, такие как круглые скобки и запятые, являются лексемами, которые не нужно отделять обобщенным пробелом.

Далее показаны примеры того, где используются пробельные символы, а где их можно опустить.

```
return0;         // НЕПРАВИЛЬНО, должно быть return 0;
return(0);       // ПРАВИЛЬНО, пробельный символ опущен
return (0);      // ПРАВИЛЬНО, используется пробельный символ
intmain();       // НЕПРАВИЛЬНО, пробельный символ опущен
int main()       // ПРАВИЛЬНО, пробельный символ опущен в скобках
int main ( )     // ПРАВИЛЬНО, пробельный символ используется в скобках
```

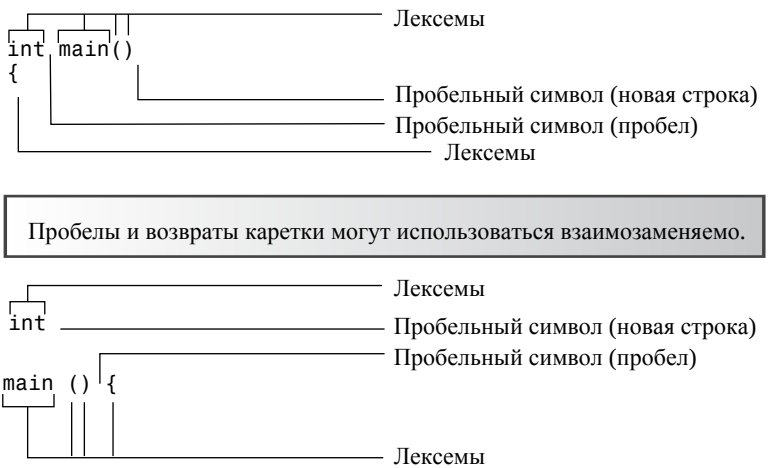


Рис. 2.3. *Лексемы и пробельные символы*

Стиль написания исходного кода C++

Хотя C++ предоставляет большую свободу в форматировании кода, программы будут легче читаться, если вы будете следовать осмысленному стилю при их написании. Допустимый, но написанный беспорядочным образом код не принесет никакого удовольствия от работы. Большинство программистов придерживаются стиля, продемонстрированного в листинге 2.1, для которого характерны перечисленные ниже правила.

- Один оператор в одной строке.
- Открывающая и закрывающая фигурные скобки для функции, каждая из которых находится в своей строке.
- Операторы в функции записаны с отступом от фигурных скобок.
- Вокруг круглых скобок, связанных с именем функции, пробельные символы отсутствуют.

Первые три правила предназначены для получения чистого и легко читаемого кода. Четвертое правило помогает отличать функции от встроенных структур C++, таких как циклы, для которых также используются круглые скобки. Далее в книге будут также описаны и другие правила.

Операторы в языке C++

Программа, написанная на языке C++, представляет собой коллекцию функций, каждая из которых, в свою очередь, является коллекцией операторов. В C++ имеется несколько разновидностей операторов, и сейчас мы рассмотрим некоторые из них. В листинге 2.2 можно увидеть операторы двух новых видов. Первый из них, *оператор объявления*, создает переменную. Второй, *оператор присваивания*, присваивает этой переменной определенное значение. Кроме этого, в программе продемонстрирована новая возможность объекта `cout`.

Листинг 2.2. `carrots.cpp`

```
// carrots.cpp -- программа по технологии производства пищевых продуктов
// использует и отображает переменную
#include <iostream>
int main()
{
    using namespace std;
    int carrots;              // объявление переменной целочисленного типа
    carrots = 25;             // присваивание значения переменной
    cout << "I have ";
    cout << carrots;          // отображение значения переменной
    cout << " carrots.";
    cout << endl;
    carrots = carrots - 1;    // изменение переменной
    cout << "Crunch, crunch. Now I have " << carrots << " carrots." << endl;
    return 0;
}
```

Раздел объявления отделен от остальной части программы пустой строкой. Этот прием часто можно было встретить в программах на C, но в программах на C++ это редкость. Далее показан результат работы программы, представленной в листинге 2.2:

```
I have 25 carrots.
Crunch, crunch. Now I have 24 carrots.
```

Следующие несколько страниц мы посвятим рассмотрению этой программы.

Операторы объявления и переменные

Компьютеры — это точные и организованные машины. Для того чтобы сохранить элемент информации в компьютере, вы должны идентифицировать как ячейку памяти, так и объем памяти, требуемый для хранения этой информации. В языке C++ проще всего это можно сделать с помощью *оператора объявления*, который идентифицирует тип памяти и предоставляет метку для ячейки. Например, в программе, представленной в листинге 2.2, имеется следующий оператор объявления (обратите внимание на наличие точки с запятой):

```
int carrots;
```

Этот оператор предоставляет два вида информации: необходимый тип хранения и метку, привязанную к этой ячейке. В частности, оператор объявляет, что программа требует объема памяти, достаточного для хранения целого числа, для которого в C++ используется метка `int`. Компилятор анализирует подробные данные о размещении

и назначении метки ячейке памяти. Компилятор C++ может обрабатывать несколько разновидностей, или типов, данных, при этом тип `int` является одним из наиболее распространенных. Он соответствует целому числу (integer) — числу без дробной части. В языке C++ тип `int` может принимать как положительные, так и отрицательные значения, а диапазон допустимых значений зависит от реализации языка. В главе 3 тип `int` и другие базовые типы данных рассматриваются более подробно.

Второй выполненной задачей является именование ячейки памяти. В этом случае оператор объявления говорит, что с этого момента программа будет использовать имя `carrots` для обозначения значения, хранящегося в указанной ячейке памяти. `carrots` — это *переменная*, поскольку ее значение можно изменять. В языке C++ все переменные должны объявляться. Если вы опустите объявление в `carrots.cpp`, компилятор выдаст сообщение об ошибке, когда программа попытается использовать переменную `carrots`. (Иногда переменную умышленно не объявляют, чтобы посмотреть, как на это отреагирует компилятор. Впоследствии, столкнувшись с подобной реакцией, вы будете знать, что нужно проверить код на предмет необъявленных переменных.)

Для чего необходимо объявлять переменные?

В некоторых языках программирования, особенно в BASIC, новая переменная создается при первом упоминании нового имени, без явного объявления. На первый взгляд, это может показаться удобным, однако у этого способа имеется большой недостаток. Если вы допустите ошибку в написании имени переменной, то непреднамеренно создадите новую переменную. То есть в BASIC может возникнуть следующая ситуация:

```
CastleDark = 34
...
CastleDank = CastleDark + MoreGhosts
...
PRINT CastleDark
```

Поскольку имя `CastleDank` написано с ошибкой (вместо `r` напечатано `n`), то изменения, которые вы произведете в этой переменной, оставят неизмененной переменную `CastleDark`. Ошибки подобного рода довольно трудно обнаружить, поскольку они не нарушают ни одного правила языка BASIC. Однако в C++ переменная `CastleDark` может быть объявлена, а написанная с ошибкой `CastleDank` — не объявлена. В результате, эквивалентный код на C++ нарушит правило обязательного объявления переменных и компилятор выдаст сообщение об ошибке.

В общем случае, объявление указывает тип сохраняемых данных и имя программы, которая будет использовать данные, хранящиеся в этой переменной. В этой конкретной ситуации программа создает переменную по имени `carrots`, в которой хранится целое число (рис. 2.4).

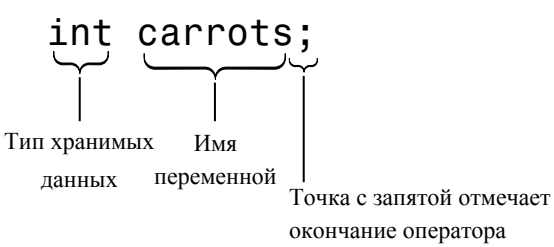


***Рис. 2.4.** Объявление переменной*

Оператор объявления в программе называется оператором *определяющего объявления*, или кратко — *определением*. Его присутствие означает, что компилятор выделит пространство памяти для хранения значений переменной. В более сложных ситуациях можно применять *ссылочное объявление*. В этом случае компьютер будет использовать переменную, которая уже где-то определена. В общем случае объявление не должно быть определением, однако, в нашем примере это так.

Если вы знакомы с языками программирования C или Pascal, то должны знать о том, как объявляются переменные. В языке C++ принят другой способ объявления переменной. Вы знаете, что в языках C и Pascal все объявления располагаются в самом начале функции или процедуры. А в языке C++ такого требования нет, поэтому объявлять переменную можно перед ее первым использованием. Таким образом, чтобы узнать о типе переменной, вам не придется возвращаться в начало программы. Пример такого объявления вы увидите далее в этой главе. Такая форма объявления имеет свой недостаток — имена всех переменных не собраны в одном месте, поэтому сразу определить, какие переменные использует функция, иногда проблематично. (Кстати говоря, в стандарте C99 для языка C правила объявления переменных такие же, как и в C++.)

Совет

В языке C++ принято объявлять переменную как можно ближе к той строке, в которой она впервые используется.

Операторы присваивания

Оператор присваивания присваивает значение ячейке памяти. Например, следующий оператор присваивает целое значение `25` ячейке памяти, обозначаемой переменной `carrots`:

```
carrots = 25;
```

Символ = называется *операцией присваивания*. Одной из необычных особенностей языка C++ (как и C) является то, что вы можете использовать операцию присваивания несколько раз подряд. Например, приведенный ниже код является вполне допустимым:

```
int steinway;
int baldwin;
int yamaha;
yamaha = baldwin = steinway = 88;
```

Операция присваивания выполняется поочередно, справа налево. Сначала значение `88` присваивается переменной `steinway`, затем это же значение присваивается переменной `baldwin` и, наконец, переменной `yamaha`. (В языке C++, как и в C, допускается написание такого причудливого кода.)

Второй оператор присваивания из листинга 2.2 демонстрирует возможность изменения значения переменной:

```
carrots = carrots - 1; // изменяет значение переменной
```

Выражение в правой части оператора присваивания (`carrots - 1`) является примером арифметического выражения. Компьютер вычитает 1 из 25, т.е. текущего значения переменной `carrots`, в результате чего получается 24. Затем операция присваивания сохраняет новое значение в ячейке `carrots`.

Новый трюк с объектом cout

До настоящего момента в примерах, приведенных в этой главе, объект cout принимал строки для последующего их отображения. В листинге 2.2 объект cout принимает переменную целочисленного типа:

```
cout << carrots;
```

Программа не выводит слово carrots; взамен этого она выводит целое значение 25, присвоенное переменной carrots. Здесь следует отметить два трюка. Во-первых, cout заменяет переменную carrots ее текущим числовым значением — 25. Во-вторых, cout транслирует это значение в соответствующие символы вывода.

Как видите, cout работает и со строками, и с целыми числами. Этот факт может показаться не особо существенным, тем не менее, учтите, что целое число 25 все же отличается от строки "25". Строка содержит символы, посредством которых вы записываете число (т.е. символ 2 и символ 5). Программа хранит числовые коды для символов 2 и 5. Для вывода строки cout печатает каждый символ. Однако целое число 25 хранится как числовое значение. Вместо того чтобы хранить каждую цифру по отдельности, компьютер хранит 25 в виде двоичного числа. (Это представление обсуждается в приложении А.) Основной смысл заключается в том, что объект cout должен преобразовать целое число в символ, прежде чем он будет выведен на экран. К тому же, cout способен определить, что переменная carrots является целочисленной и требует преобразования.

Возможно, интеллектуальность cout можно лучше понять, сравнив его со старым способом вывода в C. Чтобы напечатать строку "25" и целое число 25, в C применяется многоцелевая функция вывода printf():

```
printf("Printing a string: %s\n", "25");
printf("Printing an integer: %d\n", 25);
```

Не вдаваясь в тонкости работы функции printf(), следует отметить, что для указания отображаемой информации — строки или целого числа — должны использоваться специальные коды (%s и %d). Если вы хотите, чтобы функция printf() напечатала строку, и по ошибке передаете ей целое число, то printf() не заметит здесь ошибки. Она продолжит работу и выведет на экран бессмысленный набор символов.

Что касается объекта cout, то его интеллектуальное поведение возможно благодаря средствам объектно-ориентированного программирования C++. По сути, операция вставки (<<) в языке C++ изменяет свое поведение в зависимости от того, с данными какого типа она работает. Это пример перегрузки операций. В последующих главах, когда мы займемся рассмотрением перегрузки функций и операций, вы узнаете, как самостоятельно реализовать подобные функции.

cout и printf()

Если вы привыкли программировать на C и работали с функцией printf(), работа объекта cout может показаться странной. Конечно, можно по-прежнему пользоваться функцией printf(), особенно если у вас уже накоплен опыт работы с ней. Однако на самом деле поведение cout не выглядит более чуждым, чем поведение функции printf(), со всеми ее спецификациями преобразования. Напротив, cout обладает значительными преимуществами по сравнению с printf(). Способность распознавания типов данных объектом cout свидетельствует о его более интеллектуальном и надежном проектном решении. Кроме этого, он является *расширяемым*. Это означает, что вы можете переопределять операцию <<, чтобы объект cout мог распознавать и отображать новые типы данных. И если вам по душе широкие возможности управления, которые предлагает printf(), то вы можете добиться тех же результатов и с помощью cout (см. главу 17).

Другие операторы C++

Давайте рассмотрим еще пару примеров операторов. Программа, представленная в листинге 2.3, продолжает предыдущий пример, позволяя вводить значение во время выполнения. Для этого в ней используется объект `cin` — аналог `cout`, но предназначенный для ввода. Кроме того, в программе продемонстрирован еще один способ использования универсального объекта `cout`.

Листинг 2.3. `getinfo.cpp`

```
// getinfo.cpp -- ввод и вывод
#include <iostream>
int main()
{
        using namespace std;
        int carrots;
        cout << "How many carrots do you have?" << endl;
        cin >> carrots; // ввод C++
        cout << "Here are two more. ";
        carrots = carrots + 2;
    // следующая строка выполняет конкатенацию вывода
        cout << "Now you have" << carrots << " carrots." << endl;
        return 0;
}
```

Подгонка кода программы

Если в предшествующих листингах вы добавляли `cin.get()`, вам необходимо добавить два оператора `cin.get()` в данный листинг, чтобы сохранить вывод программы видимым на экране. Первый оператор `cin.get()` будет читать символ новой строки, генерируемый нажатием клавиши <Enter> или <Return> после набора числа, а второй заставит программу приостановиться до нажатия клавиши <Enter> или <Return> вновь.

Ниже показан пример вывода программы из листинга 2.3:

```
How many carrots do you have?
12
Here are two more. Now you have 14 carrots.
```

Эта программа обладает двумя новыми особенностями: использование `cin` для чтения клавиатурного ввода и комбинирование четырех операторов вывода в один. Давайте рассмотрим это более подробно.

Использование `cin`

Как видно по выводу программы из листинга 2.3, значение, введенное с клавиатуры (12), в конечном итоге присваивается переменной `carrots`. Это осуществляется следующим оператором:

```
cin >> carrots;
```

Взглянув на него, вы наглядно видите информационный поток из объекта `cin` в переменную `carrots`. Разумеется, существует более формальное описание этого процесса. Подобно тому, как C++ рассматривает вывод как поток символов, выходящих из программы, ввод рассматривается как поток символов, входящих в программу. Файл `iostream` определяет `cin` как объект, который представляет этот поток.

При выводе операция << вставляет символы в поток вывода. При вводе объект cin использует операцию >> для извлечения символов из потока ввода. Обычно в правой части операции указывается переменная, которой будут присваиваться извлеченные данные. (Символы << и >> были выбраны для визуальной подсказки направления потока информации.)

Подобно cout, cin является интеллектуальным объектом. Он преобразует входные данные, представляющие собой последовательность символов, введенных с клавиатуры, в форму, приемлемую для переменной, которая получает эту информацию. В нашем случае программа объявляет carrots как целочисленную переменную, поэтому входные данные преобразуются в числовую форму, используемую компьютером для хранения целых чисел.

Конкатенация с помощью cout

Еще одной новой возможностью getinfo.cpp является комбинирование четырех операторов вывода в одном. В файле iostream определена операция <<, поэтому вы можете объединить (т.е. выполнить конкатенацию) вывод следующим образом:

```
cout << "Now you have " << carrots << " carrots." << endl;
```

Такая запись позволяет объединить вывод строки и вывод целого числа в одном операторе. В результате выполнения кода вывод будет таким же, как если бы использовалась следующая запись:

```
cout << "Now you have ";
cout << carrots;
cout << " carrots";
cout << endl;
```

При желании объединенную версию можно переписать так, чтобы растянуть один оператор на четыре строки:

```
cout << "Now you have "
     << carrots
     << " carrots."
     << endl;
```

Такая запись возможна благодаря тому, что в C++ новые строки и пробелы между лексемами трактуются как взаимозаменяемые. Последний способ удобно применять, когда длина строки получается слишком большой.

Обратите внимание на еще одну деталь. Предложение

```
Now you have 14 carrots.
```

выводится в той же строке, что и

```
Here are two more.
```

Это происходит потому, что, как было сказано выше, вывод одного оператора cout следует непосредственно за выводом предыдущего оператора cout. Это справедливо даже в том случае, если между ними находятся другие операторы.

cin и cout: признак класса

Вы уже знаете об объектах cin и cout достаточно много. В этом разделе вы узнаете больше о том, что собой представляют классы. Как было сказано в главе 1, класс является одним из основных концепций объектно-ориентированного программирования (ООП) в C++.

Класс — это тип данных, определяемый пользователем. Чтобы определить класс, вы описываете, какую разновидность информации он может хранить, и какие действия разрешает выполнять над этими данными. Класс имеет такое же отношение к объекту, как тип к переменной. То есть определение класса описывает форму данных и способы их использования, тогда как объект — это сущность, созданная в соответствии с упомянутой спецификацией формы данных. Здесь можно привести такую аналогию: если класс можно рассматривать в качестве категории, например, известные актеры, то объект можно рассматривать как конкретного представителя этой категории, например, мультипликационный герой Шрэк. Продолжая эту аналогию далее, можно сказать, что представление актеров в классе может включать описания действий, связанных с классом: чтение роли, выражение печали, демонстрация угрозы, получение награды и т.п. Если перейти к терминологии ООП, то можно сказать, что класс в C++ соответствует тому, что в некоторых языках программирования называется *объектным типом*, а объект в языке C++ соответствует экземпляру объекта или переменной экземпляра.

Теперь давайте перейдем к деталям. Вспомните следующее объявление переменной:

```
int carrots;
```

В этом объявлении создается конкретная переменная (`carrots`), обладающая свойствами типа `int`. Другими словами, переменная `carrots` может хранить целое число и может использоваться в различных целях, например, для вычитания или сложения чисел. Что же касается `cout`, то про него можно сказать, что это объект, созданный для представления свойств класса `ostream`. В определении класса `ostream` (также находящемся в файле `iostream`) описана разновидность данных объекта `ostream`, а также операции, которые можно выполнять над этими данными, такие как вставка числа или строки в поток вывода. Аналогично `cin` — это объект, созданный со свойствами класса `istream`, который тоже определен в `iostream`.

На заметку!

Класс описывает все свойства типа данных, включая действия, которые могут над ним выполняться, а объект является сущностью, созданной в соответствии с этим описанием.

Вы уже знаете, что классы представляют собой определяемые пользователями типы. Тем не менее, вы — как пользователь — не создаете классы `ostream` и `istream`. Подобно функциям, которые могут быть включены в библиотеки функций, классы могут находиться в библиотеках классов. Это в полной мере относится к `ostream` и `istream`. Формально они не являются встроенными классами C++; наоборот, они представляют собой примеры классов, которые описаны в стандарте языка. Определения классов расположены в файле `iostream` и не встроены в компилятор. Эти определения можно даже изменять, хотя поступать так не рекомендуется. (Точнее, это совершенно бессмысленная затея.) Семейство классов `iostream` и связанное с ним семейство `fstream` (файловый ввод-вывод) являются единственными наборами определений классов, которые входят в состав всех ранних реализаций C++. Однако по решению комитета ANSI/ISO в стандарт было включено несколько дополнительных библиотек классов. Кроме того, во многих реализациях предлагаются дополнительные определения классов, входящие в состав пакета. Действительно привлекательной чертой C++ является наличие содержательных и полезных библиотек классов, которые позволяют разрабатывать программы для платформ Unix, Macintosh и Windows.

В описании класса указываются все действия, которые могут быть выполнены над объектами этого класса. Чтобы выполнить разрешенное действие над определенным объектом, ему отправляется сообщение. Например, если необходимо, чтобы объект `cout` отобразил строку, объекту посылается сообщение, которое, фактически, говорит: "Эй, объект! Отобрази-ка вот это". Отправить сообщение в C++ можно двумя способами. В одном из них используется метод класса, что, по сути, представляет собой вызов функции. Другой способ, который используется применительно к объектам `cin` и `cout`, заключается в переопределении операции. Таким образом, следующий оператор использует переопределенную операцию `<<` для отправки команды "отобразить сообщение" объекту `cout`:

```
cout << "I am not a crook."
```

В этом случае в сообщении содержится аргумент, представляющий собой строку, которую необходимо отобразить. (Пример показан на рис. 2.5.)

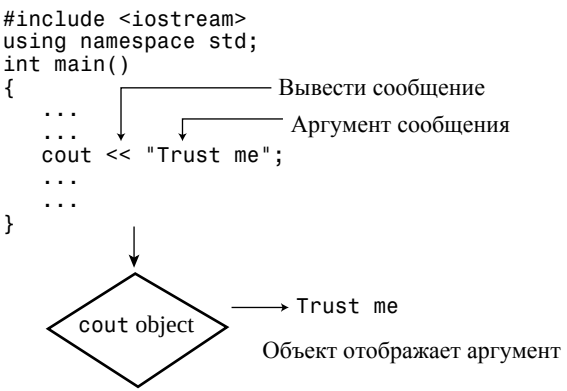


***Рис. 2.5.** Отправка сообщения объекту*

Функции

Так как функции представляют собой модули, из которых строятся программы на C++, и поскольку они необходимы для определений ООП в C++, вы должны хорошо с ними познакомиться. Ряд аспектов, связанных с функциями, являются сложными темами, поэтому основной материал о функциях будет приведен в главах 7 и 8. Тем не менее, если сейчас мы рассмотрим некоторые основополагающие характеристики функций, то при дальнейшем ознакомлении с функциями вы уже будете иметь о них некоторое представление. Итак, в оставшейся части этой главы будет дано введение в основы применения функций.

Функции в C++ можно разбить на две категории: функции, которые возвращают значения, и функции, значения не возвращающие. Для каждой разновидности функций можно найти примеры в стандартной библиотеке функций C++. Кроме того, можно создавать собственные функции обеих категорий. Давайте рассмотрим библиотечную функцию с возвращаемым значением и затем выясним, как создавать собственные простые функции.

Использование функции, имеющей возвращаемое значение

Функция, имеющая возвращаемое значение, генерирует значение, которое можно присвоить переменной или применить в каком-нибудь выражении.

Например, стандартная библиотека C/C++ содержит функцию `sqrt()`, которая возвращает квадратный корень из числа. Предположим, что требуется вычислить квадратный корень из `6.25` и присвоить его переменной `x`. В программе можно использовать следующий оператор:

```
x = sqrt(6.25); // возвращает значение 2.5 и присваивает его переменной x
```

Выражение `sqrt(6.25)` активизирует, или *вызывает*, функцию `sqrt()`. Выражение `sqrt(6.25)` называется *вызовом функции*, активизируемая функция — *вызываемой функцией*, а функция, содержащая вызов функции — *вызывающей функцией* (рис. 2.6).

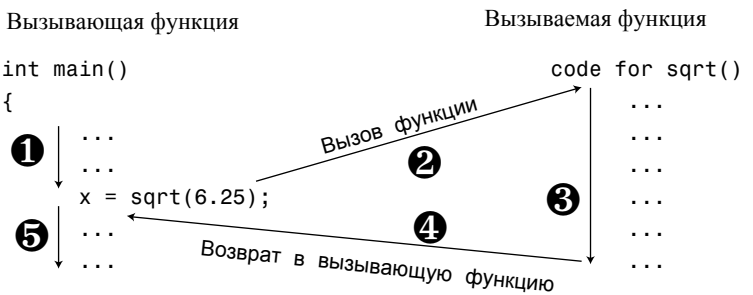


Рис. 2.6. *Вызов функции*

Значение в круглых скобках (в нашем случае — `6.25`) — это информация, которая отправляется функции; говорят, что это значение *передается* функции. Значение, которое отсылается функции подобным образом, называется *аргументом* или *параметром* (рис. 2.7). Функция `sqrt()` вычисляет ответ, равный `2.5`, и отправляет его обратно вызывающей функции; отправленное обратно значение называется *возвращаемым значением* функции. Возвращаемое значение можно представлять как значение, которое подставляется вместо вызова функции в операторе после того, как выполнение функции завершено. Так, в рассматриваемом примере возвращаемое значение присваивается переменной `x`. Говоря кратко, аргумент — это информация, которая отправляется функции, а возвращаемое значение — это значение, которое отправляется обратно из функции.

Рис. 2.6. *Синтаксис вызова функции*

Это практически все, что нужно знать, за исключением того, что прежде чем компилятор C++ сможет использовать функцию, он должен узнать, какого типа аргументы принимает функция, и к какому типу относится возвращаемое значение. То есть, возвращает ли функция целочисленное значение, или символьное, или число с дробной частью, или что-нибудь другое? При отсутствии такой информации компилятор не будет знать, как интерпретировать возвращаемое значение. Выражение этой информации в C++ осуществляется с помощью оператора прототипа функции.

На заметку!

Программа на C++ должна содержать прототипы для каждой функции, используемой в программе.

Прототип играет для функций ту же роль, что и объявление для переменных: в нем перечислены используемые типы. Например, в библиотеке C++ функция `sqrt()` определена как принимающая число с (возможно) дробной частью (например, `6.25`) в качестве аргумента и возвращающая число того же самого типа. В некоторых языках программирования такие числа называются *вещественными числами*, и в C++ для этого типа используется имя `double`. (В главе 3 тип `double` рассматривается более подробно.) Прототип функции `sqrt()` выглядит следующим образом:

```
double sqrt(double);      // прототип функции
```

Первый тип `double` означает, что функция `sqrt()` возвращает значение типа `double`. Тип `double` в круглых скобках означает, что функция `sqrt()` требует аргумента типа `double`. Таким образом, этот прототип описывает функцию `sqrt()` в точности так, как она используется в следующем фрагменте кода:

```
double x;                 // объявляет x как переменную типа double
x = sqrt(6.25);
```

Завершающая точка с запятой в прототипе идентифицирует его как оператор и поэтому назначает ему статус прототипа, а не заголовка функции. Если точку с запятой опустить, то компилятор будет интерпретировать строку как заголовок функции, и ожидать за ним тела функции, определяющего ее.

Если в программе используется функция `sqrt()`, значит, для нее должен быть предоставлен прототип. Это можно делать двумя способами:

- ввести самостоятельно прототип функции в файле исходного кода;
- воспользоваться заголовочным файлом `cmath` (`math.h` в старых системах), который содержит прототип функции.

Второй вариант лучше, поскольку заголовочный файл всегда предпочтительнее самостоятельного ввода прототипа. Каждая функция в библиотеке C++ имеет прототип в одном или нескольких заголовочных файлах. Достаточно просмотреть описание функции в руководстве или в оперативной справочной системе, где должно быть сказано, какой заголовочный файл должен использоваться. Например, в описании функции `sqrt()` сказано о необходимости применения заголовочного файла `cmath`. (Здесь опять, возможно, понадобится использовать старый заголовочный файл `math.h`, который работает для программ на C и C++.)

Не следует путать прототип функции с определением функции. Прототип, как уже было показано, описывает только интерфейс функции. Другими словами, он указывает информацию, которую получает функция, и информацию, которую она отправляет обратно. Определение, с другой стороны, включает код для обеспечения работы функции — например, код для вычисления квадратного корня числа.

Языки C и C++ разделяют эти два средства — прототип и определение — для библиотечных функций. Библиотечные файлы содержат скомпилированный код для функций, а заголовочные файлы — соответствующие прототипы.

Прототип функции должен располагаться перед первым использованием функции. Чаще всего прототипы помещают непосредственно перед определением функции `main()`. В листинге 2.4 приведен пример использования библиотечной функции `sqrt()`; ее прототип предоставляется включением файла `cmath`.

Листинг 2.4. `sqrt.cpp`

```
// sqrt.cpp -- использование функции sqrt()
#include <iostream>
#include <cmath> // или math.h
int main()
{
     using namespace std;

     double area;
     cout << "Enter the floor area, in square feet, of your home: ";
     cin >> area;
     double side;
     side = sqrt(area);
     cout << "That's the equivalent of a square " << side
          << " feet to the side." << endl;
     cout << "How fascinating!" << endl;
     return 0;
}
```

Использование библиотечных функций

Библиотечные функции C++ хранятся в библиотечных файлах. Когда компилятор компилирует программу, он должен искать библиотечные файлы для используемых функций. Компиляторы отличаются между собой тем, какие библиотечные файлы они ищут автоматически. Если вы попробуете выполнить код из листинга 2.4 и получите сообщение о том, что `_sqrt` является неопределенной внешней функцией, это значит, что компилятор не осуществляет автоматический поиск библиотеки математических функций. (Компиляторы обычно добавляют к именам функций символ подчеркивания в качестве префикса — это еще один намек о том, что они оставляют за собой последнее слово о вашей программе.) Если вы получили такое сообщение, просмотрите документацию по компилятору: в ней должно быть сказано, как заставить компилятор находить необходимую библиотеку. Например, в обычной реализации Unix может потребоваться указание опции `-lm` (*l*ibrary *m*ath) в конце командной строки:

```
CC sqrt.C -lm
```

Некоторые версии компилятора Gnu в среде Linux ведут себя аналогично:

```
g++ sqrt.C -lm
```

Простое включение заголовочного файла `cmath` предоставляет прототип, но не обязательно приводит к тому, что компилятор будет искать соответствующий библиотечный файл.

Ниже показан пример выполнения программы, представленной в листинге 2.4:

```
Enter the floor area, in square feet, of your home: 1536
That's the equivalent of a square 39.1918 feet to the side.
How fascinating!
```

Поскольку функция sqrt() работает со значениями типа double, в этом примере переменные тоже имеют тип double. Обратите внимание, что объявление переменной типа double имеет ту же форму, или синтаксис, что и объявление переменной типа int:

```
имя-типа имя-переменной;
```

Тип double позволяет переменным area и side хранить значения с десятичными частями, например, 1536.0 и 39.1918. В переменной типа double явное целое число, такое как 1536, хранится в виде вещественного числа с десятичной частью .0. Как будет показано в главе 3, тип double поддерживает более широкий диапазон значений, нежели тип int.

Язык C++ позволяет объявлять новую переменную в любом месте программы, поэтому в sqrt.cpp переменная side объявлена непосредственно перед ее использованием. Кроме того, C++ позволяет присваивать значение переменной при ее создании, поэтому допускается следующая форма записи:

```
double side = sqrt(area);
```

Такая форма называется *инициализацией*, и она подробно рассматривается в главе 3.

Обратите внимание, что объект cin знает о том, как преобразовать информацию из потока ввода в тип double, а объекту cout известно, каким образом вставить тип double в поток вывода. Как отмечалось ранее, эти объекты достаточно интеллектуальны.

Разновидности функций

Для некоторых функций требуется более одного элемента информации. Такие функции принимают несколько аргументов, разделенных запятыми. Например, математическая функция pow() принимает два аргумента и возвращает значение, равное первому аргументу, возведенному в степень, указанную во втором аргументе. Прототип этой функции выглядит следующим образом:

```
double pow(double, double);   // прототип функции с двумя аргументами
```

Если, например, необходимо вычислить 5^8 (5 в степени 8), можно воспользоваться этой функцией, как показано ниже:

```
answer = pow(5.0, 8.0);       // вызов функции со списком аргументов
```

Есть функции, которые вообще не принимают аргументов. Например, одна из библиотек C (связанная с заголовочным файлом cstdlib или stdlib.h) содержит функцию rand(), не принимающую аргументов и возвращающую случайное целое число. Ее прототип выглядит так:

```
int rand(void);               // прототип функции, не принимающей аргументов
```

Ключевое слово void явно указывает на то, что функция не принимает аргументов. Если опустить это слово и оставить скобки пустыми, C++ интерпретирует это как неявное объявление об отсутствии аргументов. Эту функцию можно использовать следующим образом:

```
myGuess = rand();             // вызов функции без аргументов
```

Обратите внимание, что в отличие от некоторых языков программирования, в C++ должны использоваться круглые скобки в вызове функции, даже если аргументы отсутствуют.

Существуют также функции, которые не имеют возвращаемого значения. Например, предположим, что вы написали функцию, которая отображает число в формате долларов и центов. Если передать ей аргумент, например, 23.5, на экране будет отображена сумма $23.50. Поскольку эта функция выводит значение на экран, а не передает его вызывающей программе, она не требует возвращаемого значения. Эта особенность функции указывается в прототипе с использованием ключевого слова void для возвращаемого типа:

```
void bucks(double); // прототип для функции, не имеющей возвращаемого значения
```

Поскольку функция bucks() не возвращает значения, ее нельзя применять в качестве части оператора присваивания или какого-то другого выражения. Вместо этого вы имеете дело с чистым оператором вызова функции:

```
bucks(1234.56);     // вызов функции; возвращаемого значения нет
```

В некоторых языках программирования термин *функция* используется только для функций, которые возвращают значения, а термин *процедура* или *подпрограмма* – для функций, не возвращающих значение. Однако в C++, как и в C, понятие *функция* применяется в отношении обеих разновидностей.

Функции, определяемые пользователем

В стандартной библиотеке C насчитывается свыше 140 предварительно определенных функций. Если какая-то из них подходит для решения вашей задачи, то, конечно же, имеет смысл ею воспользоваться. Однако нередко приходится писать собственные функции, в частности, при разработке классов. В любом случае проектировать собственные функции очень интересно, поэтому давайте рассмотрим этот процесс. Вы уже использовали некоторые предварительно определенные функции, и все они имели имя main(). Каждая программа на C++ должна иметь функцию main(), которую обязан определить пользователь. Предположим, что требуется добавить еще одну функцию, определяемую пользователем. Как и в случае с библиотечной функцией, вызов функции, определяемой пользователем, производится по ее имени. Вдобавок, как и для библиотечной функции, вы должны предоставить прототип функции перед ее использованием; обычно прототип размещается выше определения main(). Однако теперь вы, а не поставщик библиотеки, должны предоставить исходный код для новой функции. Проще всего это сделать, поместив нужный код в тот же файл после кода main(). Все сказанное проиллюстрировано в листинге 2.5.

Листинг 2.5. ourfunc.cpp

```
// ourfunc.cpp -- определение собственной функции
#include <iostream>
void simon(int);          // прототип функции simon()

int main()
{
     using namespace std;
     simon(3);            // вызов функции simon()
     cout << "Pick an integer: ";
     int count;
     cin >> count;
     simon(count);        // еще один вызов simon()
     cout << "Done!" << endl;
     return 0;
}
```

```
void simon(int n)          // определение функции simon()
{
     using namespace std;
     cout << "Simon says touch your toes " << n << " times." << endl;
} // функции void не требуют операторов return
```

Функция `main()` вызывает функцию `simon()` два раза: один раз с аргументом 3, а другой раз — с аргументом-переменной `count`. Между этими вызовами пользователь вводит целое число, которое присваивается переменной `count`. В этом примере в приглашении на ввод значения для `count` символ новой строки не используется. В результате пользователь будет вводить значение в той же строке, где располагается приглашение. Ниже показан пример выполнения этой программы:

```
Simon says touch your toes 3 times.
Pick an integer: 512
Simon says touch your toes 512 times.
Done!
```

Форма функции

Определение функции `simon()` в листинге 2.5 следует той же общей форме, что и определение `main()`. Сначала идет заголовок функции. Затем в фигурных скобках следует тело функции. Форма определения функции может быть обобщена следующим образом:

```
тип имя_функции(список_аргументов)
{
   операторы
}
```

Обратите внимание, что исходный код, определяющий функцию `simon()`, расположен после закрывающей фигурной скобки функции `main()`. Как и в C, но в отличие от Pascal, в языке C++ не допускается помещать одно определение функции внутрь другого. Каждое определение функции располагается отдельно от остальных определений; все функции создаются одинаково (рис. 2.8).

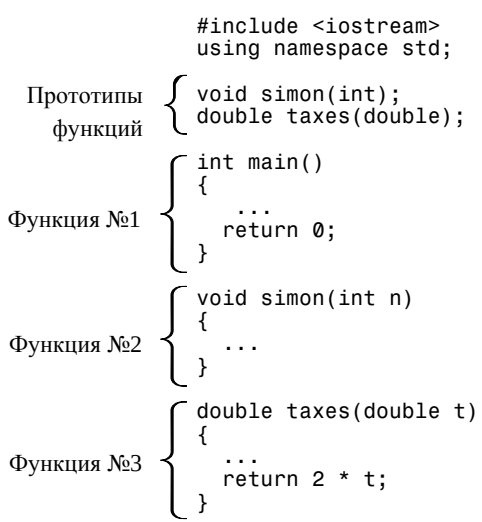


***Рис. 2.8.** Определения функций располагаются в файле последовательно*

Заголовки функций

Функция `simon()` из листинга 2.5 имеет следующий заголовок:

```
void simon(int n)
```

Здесь `void` означает, что функция `simon()` не имеет возвращаемого значения. Поэтому при ее вызове не генерируется значение, которое можно было бы присвоить какой-то переменной в `main()`. Таким образом, первый вызов функции имеет вид:

```
simon(3);              // нормально для функций void
```

Поскольку `simon()` не возвращает значения, ее нельзя использовать следующим образом:

```
simple = simon(3);   // не допускается для функций void
```

`int n` в скобках означает, что функция `simon()` будет вызываться с одним аргументом типа `int`. Здесь `n` — это новая переменная, которой было присвоено значение, переданное во время вызова функции. Таким образом, при следующем вызове функции переменной `n`, определенной в заголовке функции `simon()`, присваивается значение 3:

```
simon(3);
```

Когда оператор `cout` в теле функции использует переменную `n`, он имеет дело со значением, переданным при вызове функции. Поэтому вызов `simon(3)` отображает в своем выводе значение 3. В результате вызова `simon(count)` на экране будет отображено значение 512, поскольку это значение было введено для переменной `count`. Короче говоря, заголовок для `simon()` сообщает о том, что эта функция принимает один аргумент типа `int` и не имеет возвращаемого значения.

Давайте взглянем еще раз на заголовок функции `main()`:

```
int main()
```

Здесь `int` означает, что функция `main()` возвращает целочисленное значение. Пустые скобки (которые необязательно могут содержать `void`) означают, что эта функция не принимает аргументов. Функции, которые имеют возвращаемые значения, должны использовать ключевое слово `return` для передачи возвращаемого значения и завершения функции. Поэтому в конце функции `main()` присутствует следующий оператор:

```
return 0;
```

Логически все верно: функция `main()` должна возвращать значение типа `int`, и таковым является `0`. Но может возникнуть резонный вопрос: куда возвращается значение? Ведь ни в одной программе не встречается вызов функции `main()` наподобие:

```
squeeze = main();    // такого нет в наших программах
```

Дело в том, что здесь предполагается, что вашу программу может вызывать операционная система (к примеру, Unix или Windows). Поэтому возвращаемое функцией `main()` значение передается не в другую часть программы, а в ОС. Многие ОС могут использовать значения, возвращаемые программами. Например, для запуска программ и проверки возвращаемых ими значений, обычно называемых *значениями завершения*, могут быть написаны сценарии оболочки Unix и пакетные файлы командной строки Windows. Обычно нулевое значение завершения означает, что программа была выполнена успешно, а ненулевое значение свидетельствует об ошибке. Таким образом, можно написать программу на C++, которая будет возвращать ненулевое

значение в случае, скажем, невозможности открытия файла. После этого можно разработать сценарий оболочки или пакетный файл для запуска такой программы и принятия альтернативного действия в случае ее сбоя.

Ключевые слова

Ключевые слова формируют словарь компьютерного языка. В этой главе использовались четыре ключевых слова: `int`, `void`, `return` и `double`. Поскольку эти ключевые слова зарезервированы исключительно для нужд языка C++, вы не можете использовать их для других целей. То есть нельзя применять `return` в качестве имени переменной или `double` в качестве имени функции. Однако их можно использовать как часть имени, например `painter` или `return_aces`. В приложении Б вы найдете полный перечень ключевых слов C++. Кстати говоря, `main` — это не ключевое слово, поскольку оно не является частью языка программирования. Вместо этого `main` представляет собой имя обязательной функции. `main` можно использовать в качестве имени переменной. (Однако это может привести к возникновению специфических проблем, рассказать о которых в этой книге не представляется возможным; если коротко, то лучше не поступать так.) Подобным образом, другие имена функций и объектов не являются ключевыми словами. Однако применение одного и того же имени, например `cout`, для объекта и для переменной в программе приведет к ошибке компиляции. Другими словами, `cout` можно использовать как имя переменной в функции, в которой не задействован объект `cout` для вывода, но в рамках одной функции нельзя применять `cout` для имени переменной и объекта вывода.

Использование определяемых пользователем функций, имеющих возвращаемое значение

Давайте продолжим рассмотрение функций и перейдем к тем из них, в которых используется оператор возврата. На примере `main()` уже был продемонстрирован проект функции, имеющей возвращаемое значение: определить возвращаемый тип в заголовке функции и предоставить оператор `return` в конце тела функции. Эту форму можно использовать, чтобы решить задачу с весом для иностранцев, приезжающих в Великобританию. В Великобритании используются весы, мерой которых являются *стоуны* (1 стоун равен 14 фунтам, или приблизительно 6,34 кг), а в США применяются фунты или килограммы. В программе, представленной в листинге 2.6, для преобразования стоунов в фунты используется функция.

Листинг 2.6. `convert.cpp`

```
// convert.cpp -- преобразует стоуны в фунты
#include <iostream>
int stonetolb(int); // прототип функции
int main()
{
   using namespace std;
   int stone;
   cout << "Enter the weight in stone: ";
   cin >> stone;
   int pounds = stonetolb(stone);
   cout << stone << " stone = ";
   cout << pounds << " pounds." << endl;
   return 0;
}
int stonetolb(int sts)
{
   return 14 * sts;
}
```

Ниже показан пример выполнения этой программы:

```
Enter the weight in stone: 15
15 stone = 210 pounds.
```

В функции `main()` программа использует объект `cin` для ввода значения целочисленной переменной `stone`. Это значение передается функции `stonetolb()` в качестве аргумента и присваивается переменной `sts` в этой функции. Функция `stonetolb()` использует ключевое слово `return` для возврата значения `14*sts` в функцию `main()`. Это пример того, что после оператора `return` не обязательно должно следовать простое число. В этом случае с помощью выражения вы избавляетесь от необходимости создавать новую переменную, которой должно быть присвоено значение, прежде чем оно будет возвращено. Программа вычисляет значение этого выражения (в нашем примере — `210`) и возвращает его. Для возврата значения можно использовать и более громоздкую форму записи:

```
int stonetolb(int sts)
{
   int pounds = 14 * sts;
   return pounds;
}
```

Оба варианта дают один и тот же результат. Второй вариант с разделением вычислений и возврата проще читать и модифицировать.

В общем случае функцию с возвращаемым значением можно использовать там, где применяется простая константа того же самого типа. Например, функция `stonetolb()` возвращает значение типа `int`. Это значит, что `stonetolb()` можно использовать следующим образом:

```
int aunt = stonetolb(20);
int aunts = aunt + stonetolb(10);
cout << "Ferdie weighs " << stonetolb(16) << " pounds." << endl;
```

В каждом из случаев программа вычисляет возвращаемое значение и затем использует его в этих операторах.

Как можно было видеть в приведенных примерах, прототип функции описывает ее интерфейс, т.е. способ взаимодействия функции с остальной частью программы. Список аргументов показывает, какой тип информации передается функции, а тип функции указывает на тип возвращаемого ею значения. Программисты иногда представляют функции как *черные ящики* (термин, заимствованный из электроники), для которых известны только входящий и исходящий потоки информации. Эту точку зрения лучше всего отражает прототип функции (рис. 2.9).

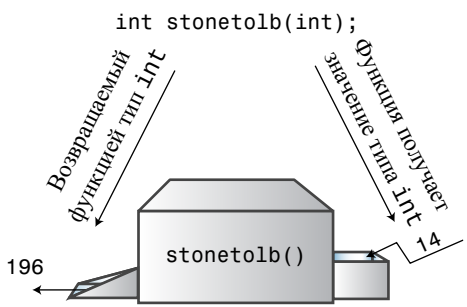


Рис. 2.9. Прототип функции и функция как черный ящик

Несмотря на свою краткость и простоту, stonetolb() обладает полным набором функциональных возможностей:

- имеет заголовок и тело;
- принимает аргумент;
- возвращает значение;
- требует прототип.

Функцию stonetolb() можно брать за основу при проектировании других функций. В главах 7 и 8 мы продолжим изучение функций. А материал в этой главе должен позволить вам понять работу функций в C++.

Местоположение директивы using в программах с множеством функций

Обратите внимание, что в листинге 2.5 директива using присутствует в каждой из двух функций:

```
using namespace std;
```

Это объясняется тем, что каждая функция использует объект cout и потому должна иметь доступ к определению cout в пространстве имен std.

Еще один способ сделать пространство имен std доступным для обеих функций в листинге 2.5 предусматривает размещение директивы за пределами функций – перед ними:

```
// ourfunc1.cpp -- изменение местоположения директивы using
#include <iostream>
using namespace std; // влияет на все определения функций в этом файле
void simon(int);

int main()
{
   simon(3);
   cout << "Pick an integer: ";
   int count;
   cin >> count;
   simon(count);
   cout << "Done!" << endl;
   return 0;
}

void simon(int n)
{
   cout << "Simon says touch your toes " << n << " times." << endl;
}
```

Предпочтительнее поступать так, чтобы доступ к пространству имен std был предоставлен только тем функциям, которым он действительно необходим. Например, в листинге 2.6 объект cout используется только в функции main(), поэтому нет необходимости делать доступным пространство имен std для функции stonetolb(). Таким образом, директива using помещается внутри одной лишь функции main(), предоставляя доступ к пространству имен std только этой функции.

Итак, сделать доступным пространство имен std для программы можно несколькими способами, которые перечислены ниже.

- Можно поместить следующий оператор `using` перед определением функции в файле, в результате чего все содержимое пространства имен `std` будет доступно для каждой функции в файле:

  ```
  using std namespace;
  ```

- Можно поместить следующий оператор `using` в определение функции, в результате чего для этой функции будет доступно все содержимое пространства имен `std`:

  ```
  using std namespace;
  ```

- Вместо того чтобы использовать

  ```
  using std namespace;
  ```

  можно поместить объявления `using` наподобие следующего в определение функции и сделать доступным для каждой функции конкретный элемент, такой как `cout`:

  ```
  using std::cout;
  ```

- Можно опустить директивы `using` и объявления полностью и указывать префикс `std::` всякий раз при использовании элементов из пространства имен `std`:

  ```
  std::cout << "I'm using cout and endl from the std namespace" << std::endl;
  ```

Соглашения об именовании

У программистов на C++ есть возможность использовать различные способы назначения имен функциям, классам и переменным. Среди программистов бытуют строгие и разнообразные мнения относительно стиля написания кода, что нередко служит поводом для серьезных словесных баталий на открытых форумах. Если говорить о возможных вариантах именования функций, то можно избрать любой из следующих стилей:

```
MyFunction()
myfunction()
myFunction()
my_function()
my_funct()
```

Выбор зависит от команды разработчиков, индивидуальных характеристик используемых методов или библиотек, а также от предпочтений конкретного программиста. Существует мнение, что допустим любой стиль, который не противоречит правилам C++, представленным в главе 3, поэтому решение всецело зависит от вас.

Если отстраниться от правил языка программирования, следует отметить, что выработка личного стиля именования, способствующего согласованности и точности записи, заслуживает всяческого уважения. Если программист выработает аккуратный и узнаваемый стиль именования, то это не только будет служить признаком того, что программы, написанные им, являются качественными, но и может помочь в программистской карьере.

Резюме

Программа на языке C++ состоит из одного или нескольких модулей, называемых функциями. Выполнение программы начинается с функции по имени `main()` (все символы — в нижнем регистре), поэтому любая программа должна обязательно включать эту функцию. Функция состоит из заголовка и тела. В заголовке функции указывается, каким является возвращаемое значение (если оно предусмотрено),

генерируемое функцией, и какую информацию принимает функция в аргументах. Тело функции состоит из последовательности операторов языка C++, заключенных в фигурные скобки ({}).

В C++ выделяют следующие типы операторов.

- **Оператор объявления**. В операторе объявления указывается имя и тип переменной, которая используется в функции.
- **Оператор присваивания**. Этот оператор использует операцию присваивания (=) для установки значения переменной.
- **Оператор сообщения**. Оператор сообщения посылает сообщение объекту, инициируя некоторое действие.
- **Вызов функции**. Вызов функции активизирует ее. Когда вызываемая функция завершает свою работу, программа возвращается к оператору в вызывающей функции, который непосредственно следует за вызовом функции.
- **Прототип функции**. В прототипе функции объявляется тип возвращаемого функцией значения, а также количество и типы аргументов, передаваемых функции.
- **Оператор возврата**. Оператор возврата посылает значение из вызываемой функции обратно вызывающей функции.

Класс — это определяемая пользователем спецификация типа данных. В ней подробно описан способ представления информации и действия, которые могут выполняться над этими данными. Объект — это сущность, созданная в соответствии с предписанием класса, а простая переменная является сущностью, созданной в соответствии с описанием типа данных.

В языке C++ имеются два предварительно определенных объекта (`cin` и `cout`) для обработки ввода и вывода. Они являются примерами классов `istream` и `ostream`, которые определены в файле `iostream`. Эти классы рассматривают ввод и вывод как поток символов. Операция вставки (<<), которая определена для класса `ostream`, позволяет помещать данные в поток вывода, а операция извлечения (>>), которая определена для класса `istream`, позволяет извлекать информацию из потока ввода. Как `cin`, так и `cout` являются интеллектуальными объектами, способными автоматически преобразовывать информацию из одной формы в другую в соответствии с контекстом программы.

В C++ можно пользоваться обширным набором библиотечных функций C. Чтобы работать с библиотечной функцией, необходимо включить заголовочный файл, который предоставляет прототип для функции.

Теперь, когда вы уже знаете, что собой представляют простые программы на языке C++, можно приступить к изучению следующих глав, вникая во все более широкий круг вопросов.

Вопросы для самоконтроля

Ответы на вопросы для самоконтроля по каждой главе можно найти в приложении К.

1. Как называются модули программ, написанных на языке C++?
2. Что делает следующая директива препроцессора?

```
#include <iostream>
```

3. Что делает следующий оператор?

```
using namespace std;
```

4. Какой оператор вы могли бы использовать, чтобы напечатать фразу "Hello, world" и перейти на новую строку?
5. Какой оператор вы могли бы использовать для создания переменной целочисленного типа по имени cheeses?
6. Какой оператор вы могли бы использовать, чтобы присвоить переменной cheeses значение 32?
7. Какой оператор вы могли бы использовать, чтобы прочитать значение, введенное с клавиатуры, и присвоить его переменной cheeses?
8. Какой оператор вы могли бы использовать, чтобы напечатать фразу "We have X varieties of cheese", в которой X заменяется текущим значением переменной cheeses?
9. Что говорят вам о функциях следующие прототипы?

```
int froop(double t);
void rattle(int n);
int prune(void);
```

10. В каких случаях не используется ключевое слово return при определении функции?
11. Предположим, что функция main() содержит следующую строку:

```
cout << "Please enter your PIN: ";
```

Также предположим, что компилятор сообщил о том, что cout является неизвестным идентификатором. Какова вероятная причина этого сообщения, и как выглядят способы решения проблемы?

Упражнения по программированию

1. Напишите программу на C++, которая отобразит ваше имя и адрес (можете указать фиктивные данные).
2. Напишите программу на C++, которая выдает запрос на ввод расстояния в фарлонгах и преобразует его в ярды. (Один фарлонг равен 220 ярдам, или 201 168 м.)
3. Напишите программу на C++, которая использует три определяемых пользователем функции (включая main()) и генерирует следующий вывод:

```
Three blind mice
Three blind mice
See how they run
See how they run
```

Одна функция, вызываемая два раза, должна генерировать первые две строки, а другая, также вызываемая два раза — оставшиеся строки.

4. Напишите программу, которая предлагает пользователю ввести свой возраст. Затем программа должна отобразить возраст в месяцах:

```
Enter your age: 29
Your age in months is 348.
```

5. Напишите программу, в которой функция `main()` вызывает определяемую пользователем функцию, принимающую в качестве аргумента значение температуры по Цельсию и возвращающую эквивалентное значение температуры по Фаренгейту. Программа должна выдать запрос на ввод значения по Цельсию и отобразить следующий результат:

```
Please enter a Celsius value: 20
20 degrees Celsius is 68 degrees Fahrenheit.
```

Вот формула для этого преобразования:

Температура в градусах по Фаренгейту =
1,8 * *Температура в градусах по Цельсию* + 32

6. Напишите программу, в которой функция `main()` вызывает определяемую пользователем функцию, принимающую в качестве аргумента расстояние в световых годах и возвращающую расстояние в астрономических единицах. Программа должна выдать запрос на ввод значения в световых годах и отобразить следующий результат:

```
Enter the number of light years: 4.2
4.2 light years = 265608 astronomical units.
```

Астрономическая единица равна среднему расстоянию Земли от Солнца (около 150 000 000 км, или 93 000 000 миль), а световой год соответствует расстоянию, пройденному лучом света за один земной год (примерно 10 триллионов километров, или 6 триллионов миль). (Ближайшая звезда после Солнца находится на расстоянии 4,2 световых года.) Используйте тип `double` (как в листинге 2.4) и следующий коэффициент преобразования:

1 *световой год* = 63 240 *астрономических единиц*

7. Напишите программу, которая выдает запрос на ввод значений часов и минут. Функция `main()` должна передать эти два значения функции, имеющей тип `void`, которая отобразит эти два значения в следующем виде:

```
Enter the number of hours: 9
Enter the number of minutes: 28
Time: 9:28
```

3

Работа с данными

В ЭТОЙ ГЛАВЕ...

- Правила назначения имен переменным C++
- Встроенные целочисленные типы C++: `unsigned long`, `long`, `unsigned int`, `int`, `unsigned short`, `short`, `char`, `unsigned char`, `signed char`, `bool`
- Дополнения C++11: `unsigned long long` и `long long`
- Файл `climits`, представляющий системные ограничения для различных целочисленных типов
- Числовые литералы (константы) различных целочисленных типов
- Использование квалификатора `const` для создания символьных констант
- Встроенные типы с плавающей точкой C++: `float`, `double` и `long double`
- Файл `cfloat`, представляющий системные ограничения для различных типов с плавающей точкой
- Числовые литералы различных типов с плавающей точкой
- Арифметические операции C++
- Автоматические преобразования типов
- Принудительные преобразования типов (приведения типов)

Сущность объектно-ориентированного программирования (ООП) заключается в проектировании и расширении собственных типов данных. Проектирование собственных типов данных — это усилия по приведению типов в соответствие с данными. Если вам удастся справиться с этой задачей, то в дальнейшем работать с данными будет намного удобнее. Однако прежде чем создавать собственные типы данных, вы должны изучить встроенные типы данных в C++, поскольку именно они будут выступать в качестве строительных блоков.

Встроенные типы данных C++ разделены на две группы: фундаментальные типы и составные типы. В этой главе речь пойдет об фундаментальных типах, которые представляют целые числа и числа с плавающей точкой. На первый взгляд может показаться, что для представления этих чисел можно обойтись всего двумя типами. Однако это не так: в языке C++ считается, что один целочисленный тип и один тип с плавающей точкой не в состоянии удовлетворить всем требованиям программистов, поэтому для каждого из этих типов доступно множество вариантов. В главе 4 мы рассмотрим некоторые типы, построенные на основе базовых типов; к этим дополнительным составным типам относятся массивы, строки, указатели и структуры.

Естественно, программа должна иметь способ идентификации хранящихся данных. В настоящей главе рассматривается один из методов для этого — использование переменных. Затем будет показано, как реализуются арифметические действия в C++, и, наконец — как в C++ осуществляется преобразование значений из одного типа в другой.

Простые переменные

Как правило, программа должна хранить разнообразную информацию — например, текущую цену на акции Google, величину средней влажности в Нью-Йорке в августе месяце, самую часто встречающуюся букву в тексте Конституции США и частоту ее употребления или количество современных пародистов Элвиса Пресли. Чтобы сохранить элемент информации в памяти компьютера, программа должна отслеживать три фундаментальных свойства:

- где хранится информация;
- какое значение сохранено;
- разновидность сохраненной информации.

В примерах, приведенных до сих пор, использовалась стратегия объявления переменной. Тип, указываемый при объявлении, описывает разновидность информации, а имя переменной является символическим представлением значения. Предположим, что ассистент заведующего лабораторией использует следующие операторы:

```
int braincount;
braincount = 5;
```

Эти операторы сообщают программе, что она хранит целое число, а имя `braincount` представляет целочисленное значение, в данном случае 5. По существу программа выделяет некоторую область памяти, способную уместить целое число, отмечает ее расположение и копирует туда значение 5. В дальнейшем для доступа к этому расположению можно использовать `braincount`. Эти операторы ничего не говорят о том, где именно в памяти хранится значение, но программа отслеживает эту информацию. Разумеется, с помощью операции `&` можно получить адрес `braincount` в памяти. Об этой операции речь пойдет в следующей главе, когда будет рассматриваться еще одна стратегия идентификации данных — использование указателей.

Имена, назначаемые переменным

В C++ приветствуется назначение переменным осмысленных имен. Если переменная представляет стоимость поездки, то для нее следует выбрать такое имя, как `cost_of_trip` или `costOfTrip`, но не `x` или `cot`. В C++ необходимо придерживаться следующих простых правил именования.

- В именах разрешено использовать только алфавитных символов, цифр и символа подчеркивания (`_`).
- Первым символом имени не должна быть цифра.
- Символы в верхнем и нижнем регистре рассматриваются как разные.
- В качестве имени нельзя использовать ключевое слово C++.
- Имена, которые начинаются с двух символов подчеркивания или с одного подчеркивания и следующей за ним буквы в верхнем регистре, зарезервированы для использования реализациями C++, т.е. с ними имеют дело компиляторы и ресурсы. Имена, начинающиеся с одного символа подчеркивания, зарезервированы для применения в качестве глобальных идентификаторов в реализациях.
- На длину имени не накладывается никаких ограничений, и все символы в имени являются значащими. Однако некоторые платформы могут вводить свои ограничения на длину.

Предпоследнее правило немного отличается от предыдущих, поскольку использование такого имени, как `__time_stop` или `_Donut`, не вызывает ошибку компиляции; вместо этого оно приводит к неопределенному поведению. Другими словами, в этом случае невозможно сказать, каким будет конечный результат. Причина отсутствия ошибки компиляции в том, что указанные имена не являются недопустимыми, а зарезервированными для применения в реализации. Вопрос о глобальных именах связан с тем, в каком месте программы они объявлены; об этом речь пойдет в главе 4.

Последнее правило отличает C++ от стандарта ANSI C (C99), который утверждает, что значащими являются только первые 63 символа имени. (В ANSI C два имени с одинаковыми 63 символами в начале считаются одинаковыми, даже если 64-е символы у них разные.)

Ниже приведены примеры допустимых и недопустимых имен в C++:

```
int poodle;    // допустимое
int Poodle;    // допустимое и отличающееся от poodle
int POODLE;    // допустимое и отличающееся от двух предыдущих
Int terrier;   // недопустимое – должно использоваться int, а не Int
int my_stars3  // допустимое
int _Mystars3; // допустимое, но зарезервированное – начинается с
подчеркивания
int 4ever;     // недопустимое, потому что начинается с цифры
int double;    // недопустимое – double является ключевым словом C++
int begin;     // допустимое – begin является ключевым словом Pascal
int __fools;   // допустимое, но зарезервированное – начинается с двух
подчеркиваний
int the_very_best_variable_i_can_be_version_112;  // допустимое
int honky-tonk;       // недопустимое – дефисы не разрешены
```

Обычно если переменной назначается имя, состоящее из двух или более слов, то для разделения слов используется символ подчеркивания, как в `my_onions`, или же первые буквы всех слов, кроме первого, записываются в верхнем регистре, как

в myEyeTooth. (Ветераны языка C предпочитают применять символ подчеркивания, а приверженцы традиций Pascal — заглавные буквы.) В любом случае, каждая форма записи упрощает визуальное восприятие отдельных слов и помогает различать такие имена, как carDrip и cardRip или boat_sport и boats_port.

Схемы именования

Вокруг схем именования переменных, как и схем именования функций, ведутся горячие дискуссии, и мнения программистов по этому поводу весьма расходятся. Как и в случае с именами функций, компилятору C++ все равно, какие имена вы назначаете своим переменным — лишь бы они удовлетворяли существующим правилам. Однако выработка собственного непротиворечивого и аккуратного стиля именования принесет немалую пользу.

Как и в случае назначения имен функциям, ключевым моментом именования переменных является использование заглавных букв (см. врезку "Соглашения об именовании" в главе 2). Однако многие программисты включают в имя переменной дополнительный уровень информации — префикс, который характеризует тип переменной или ее содержимое. Например, целочисленную переменную myWeight можно назвать nMyWeight; здесь префикс n используется для представления целочисленного значения — это полезно при просмотре кода, когда описание переменной находится слишком далеко от текущей строки. В качестве альтернативы эта переменная может иметь имя intMyWeight, которое описывает ее более точно, хотя и содержит большее количество символов (истинное мучение для определенного круга программистов). Подобным образом применяются и другие префиксы: str или sz можно использовать для представления строки символов с завершающим нулем, b может представлять булевское (b от слова "Boolean") значение, p — указатель (от слова "pointer"), c — одиночный символ (от слова "character").

По мере изучения языка C++ вы будете встречать множество примеров применения префиксов в именах переменных (включая представительный префикс m_lpctstr — значение члена класса, которое содержит длинный указатель на константную строку символов с завершающим нулем), среди которых немного причудливые и, возможно, непонятные стили. Как и во всех других связанных со стилем субъективных сторонах C++, лучше всего придерживаться непротиворечивости и точности. Старайтесь назначать такие имена, которые отвечали бы вашим требованиям, предпочтениям и персональному стилю. (Или же выбирайте имена, которые удовлетворяют требованиям и стилю вашего работодателя.)

Целочисленные типы

Целыми являются числа без дробной части, например 2, 98, –5286 и 0. Целых чисел очень много, и в конечном объеме памяти компьютера нельзя представить все возможные целые числа. Таким образом, язык может представить только подмножество целых чисел. Некоторые языки предлагают только один целочисленный тип (один тип, который охватывает все!), но C++ предоставляет несколько вариантов. Благодаря этому вы можете выбрать такой целочисленный тип, который будет лучше всего отвечать конкретным требованиям программы. Проблема соответствия типа данным положена в основу проектирования типов данных в ООП.

Разнообразные целочисленные типы в C++ отличаются друг от друга объемом памяти, выделяемой для хранения целого значения. Чем больше объем памяти, тем шире диапазон представляемых целочисленных значений. Вдобавок некоторые типы (со знаком) могут представлять как положительные, так и отрицательные значения, а другие (без знака) — только положительные. Обычно для описания объема памяти, используемого для хранения целого числа, применяется термин *ширина*. Чем больше памяти необходимо для хранения значения, тем оно шире. В C++ базовыми целочисленными типами, в порядке увеличения ширины, являются char, short, int, long и

появившийся в C++11 тип long long. Каждый из этих типов имеет версии со знаком и без знака. Таким образом, на выбор предлагается десять различных целочисленных типов. Давайте рассмотрим их более детально. Поскольку тип char имеет ряд специальных свойств (чаще всего он используется для представления символов, а не чисел), мы сначала опишем остальные типы.

Целочисленные типы short, int, long и long long

Память компьютера состоит из единиц, называемых *битами* (см. врезку "Биты и байты" далее в этой главе). Используя различное количество битов для хранения значений, типы short, int, long и long long в C++ могут представлять до четырех различных целочисленных ширин. Было бы удобно, если бы каждый тип во всех системах всегда имел некоторую определенную ширину, например, short — 16 битов, int — 32 бита и т.д. Однако в реальности не все так просто. Ни один из вариантов не может быть подходящим для всех разработчиков. В C++ предлагается гибкий стандарт, позаимствованный у C, с некоторыми гарантированными минимальными размерами типов:

- целочисленный тип short имеет ширину не менее 16 битов;
- целочисленный тип int как минимум такой же, как short;
- целочисленный тип long имеет ширину не менее 32 битов и как минимум такой же, как int;
- целочисленный тип long long имеет ширину не менее 64 битов и как минимум такой же, как long.

Биты и байты

Фундаментальной единицей памяти компьютера является *бит*. Его можно рассматривать как электронный переключатель, который может быть установлен в одно из двух положений: "включен" и "выключен". Положение "выключен" представляет значение 0, а положение "включен" — значение 1. Порция памяти, состоящая из 8 битов, может представлять одну из 256 различных комбинаций. Число 256 получено исходя из того, что каждый бит имеет два возможных состояния, что в конечном итоге дает общую сумму комбинаций для 8 бит: 2 × 2 × 2 × 2 × 2 × 2 × 2 × 2, или 256. Таким образом, 8-битная структура может представлять значения от 0 до 255 или от –128 до 127. Каждый дополнительный бит удваивает количество комбинаций. Это означает, что 16-битная порция памяти позволяет представлять 65 536 разных значений, 32-битная — 4 294 672 296, а 64-битная — 18 446 744 073 709 551 616. Для сравнения: с помощью типа long без знака не удастся представить количество людей на земле или число звезд в нашей галактике, но посредством типа long long это вполне возможно.

Байт обычно означает 8-битную порцию памяти. В этом смысле байт представляет собой единицу измерения, которая описывает объем памяти в компьютере, причем 1 Кбайт составляет 1024 байта, а 1 Мбайт — 1024 Кбайт. Однако в C++ байт определен по-другому. *Байт C++* состоит из минимального количества смежных битов, достаточного для того, чтобы вместить базовый набор символов для реализации. Другими словами, количество возможных значений должно быть равно или превышать число индивидуальных символов. В США базовыми наборами символов обычно являются ASCII и EBCDIC, каждый из которых может быть представлен 8 битами, поэтому байт C++ обычно равен 8 битам в системах, использующих упомянутые наборы символов. С другой стороны, в интернациональном программировании могут использоваться намного большие наборы символов, такие как Unicode, поэтому в некоторых реализациях могут использоваться 16-битные или даже 32-битные байты. Для обозначения 8-битного байта иногда используется термин *октет*.

В настоящее время во многих системах используется минимальная гарантия: тип short имеет 16 битов, а тип long — 32 бита. Это по-прежнему оставляет несколько вариантов для типа int. Он может иметь ширину в 16, 24 или 32 бита и соответствовать стандарту. Он может быть даже 64 бита, предоставляя минимальную ширину long и long long. Тип int обычно имеет 16 битов (столько же, сколько и short) в старых реализациях для IBM-совместимых ПК и 32 бита (столько же, сколько и long) для Windows XP, Windows Vista, Windows 7, Mac OS X, VAX и многих других реализаций для миникомпьютеров. В некоторых реализациях можно выбирать способ обработки типа int. (Что использует ваша реализация? В следующем примере показано, как узнать ограничения для вашей системы, не обращаясь к справочному руководству.) Различия в ширинах типов между реализациями могут привести к возникновению ошибок при переносе программы на C++ из одной среды в другую, а также при использовании другого компилятора в той же самой системе. Однако эту проблему совсем несложно свести к минимуму, о чем говорится далее в главе.

Для объявления переменных будут применяться следующие имена типов:

```
short score;          // создает целочисленную переменную типа short
int temperature;      // создает целочисленную переменную типа int
long position;        // создает целочисленную переменную типа long
```

В действительности, тип short имеет более узкий диапазон значений, чем тип short int, а тип long — более узкий диапазон, чем long int, однако вряд ли что-то будет использовать более широкие формы.

Четыре типа — int, short, long и long long — являются типами со знаком, т.е. они могут разбивать свой диапазон почти пополам между положительными и отрицательными значениями. Например, диапазон значений для 16-битного int простирается от –32 768 до 32 767.

Если вы хотите узнать, какой размер отведен для целых чисел в вашей системе, можете воспользоваться средствами C++. Во-первых, можно применить операцию sizeof, которая возвращает размер типа или переменной в байтах. (*Операция* — это действие, выполняемое над одним или несколькими элементами для получения значения. Например, операция сложения, обозначаемая знаком +, суммирует два числа.) Вспомните, что смысл байта зависит от реализации, поэтому двухбайтный int может занимать 16 битов в одной системе и 32 бита в другой. Во-вторых, заголовочный файл climits (или заголовочный файл limits.h в старых реализациях) содержит информацию о системных ограничениях в отношении целого типа. В частности, в нем определены символические имена для представления различных ограничений. Например, в нем определено имя INT_MAX как наибольшее из возможных значений int и CHAR_BIT — как количество битов в байте. В листинге 3.1 показан пример использования этих средств и пример *инициализации*, которая представляет собой использование оператора объявления для присваивания значения переменной.

Листинг 3.1. limits.cpp

```
// limits.cpp -- некоторые ограничения целых чисел
#include <iostream>
#include <climits> // используйте заголовочный файл limits.h для старых систем
int main()
{
   using namespace std;
   int n_int = INT_MAX;       // инициализация n_int максимальным значением int
   short n_short = SHRT_MAX;  // символы, определенные в файле climits
   long n_long = LONG_MAX;
   long long n_llong = LLONG_MAX;
```

```
    // Операция sizeof выдает размер типа или переменной
    cout << "int is " << sizeof (int) << " bytes." << endl;
    cout << "short is " << sizeof n_short << " bytes." << endl;
    cout << "long is " << sizeof n_long << " bytes." << endl << endl;
    cout << "long long is " << sizeof n_llong << " bytes." << endl << endl;
    cout << endl;

    cout << "Maximum values:" << endl;
    cout << "int: " << n_int << endl;
    cout << "short: " << n_short << endl;
    cout << "long: " << n_long << endl;
    cout << "long long: " << n_llong << endl << endl;

    cout << "Minimum int value = " << INT_MIN << endl;
    cout << "Bits per byte = " << CHAR_BIT << endl;
    return 0;
}
```

На заметку!

Если ваша система не поддерживает тип `long long`, удалите строки, в которых используется этот тип.

Ниже показан результат выполнения программы из листинга 3.1:

```
int is 4 bytes.
short is 2 bytes.
long is 4 bytes.
long long is 8 bytes.

Maximum values:
int: 2147483647
short: 32767
long: 2147483647
long long: 9223372036854775807

Minimum int value = -2147483648
Bits per byte = 8
```

Эти конкретные значения были получены в системе, функционирующей под управлением 64-разрядной версии ОС Windows 7.

В последующих разделах мы рассмотрим наиболее важные особенности этой программы.

Операция `sizeof` и заголовочный файл `climits`

Операция `sizeof` сообщает о том, что тип `int` занимает 4 байта в базовой системе, в которой используются 8-битные байты. Операцию `sizeof` можно применять к имени типа или к имени переменной. Если эта операция выполняется над именем типа, таким как `int`, то это имя должно быть заключено в круглые скобки. Для имен переменных вроде `n_short` скобки необязательны:

```
cout << "int is " << sizeof (int) << " bytes.\n";
cout << "short is " << sizeof n_short << " bytes.\n";
```

Заголовочный файл `climits` определяет символические константы (см. врезку "Символические константы как средство препроцессора" далее в этой главе) для представления ограничений типов. Как упоминалось ранее, `INT_MAX` представляет наибольшее значение, которое может хранить тип `int`; для нашей системы Windows 7 таким значением является 2 147 483 647. Производитель компилятора предоставляет

файл climits, в котором отражены значения, соответствующие данному компилятору. Например, файл climits для ряда старых систем, в которых используются 16-битный тип int, определяет константу INT_MAX равной 22 767. В табл. 3.1 перечислены символические константы, определенные в файле climits; часть из них относится к типам, которые еще предстоит рассмотреть.

Таблица 3.1. Символические константы, определенные в файле climits

Символическая константа	Ее представление
CHAR_BIT	Количество битов в char
CHAR_MAX	Максимальное значение char
CHAR_MIN	Минимальное значение char
SCHAR_MAX	Максимальное значение signed char
SCHAR_MIN	Минимальное значение signed char
UCHAR_MAX	Максимальное значение unsigned char
SHRT_MAX	Максимальное значение short
SHRT_MIN	Минимальное значение short
USHRT_MAX	Максимальное значение unsigned short
INT_MAX	Максимальное значение int
INT_MIN	Минимальное значение int
UINT_MAX	Максимальное значение unsigned int
LONG_MAX	Максимальное значение long
LONG_MIN	Минимальное значение long
ULONG_MAX	Максимальное значение unsigned long
LLONG_MAX	Максимальное значение long long
LLONG_MIN	Минимальное значение long long
ULLONG_MAX	Максимальное значение unsigned long long

Символические константы как средство препроцессора

В файле climits содержатся строки, подобные следующей:

```
#define INT_MAX 32767
```

Вспомните, что в процессе компиляции исходный код передается сначала препроцессору. Здесь #define, как и #include, является директивой препроцессора. Эта директива сообщает препроцессору следующее: найти в программе экземпляры символической константы INT_MAX и заменить каждое вхождение значением 32767. Таким образом, директива #define работает подобно команде глобального поиска и замены в текстовом редакторе. После произведенных замен происходит компиляция измененной программы. Препроцессор ищет независимые лексемы (отдельные слова) и пропускает вложенные слова. Это значит, что он не заменяет PINT_MAXIM на P32767IM. Директиву #define можно также использовать для определения собственных символических констант (см. листинг 3.2). Однако директива #define является пережитком языка C. В C++ имеется более удобный способ создания символических констант (с помощью ключевого слова const, которое рассматривается чуть позже), поэтому применять директиву #define доведется редко. Тем не менее, она будет встречаться в ряде заголовочных файлов, особенно в тех, которые предназначены для использования в C и C++.

Инициализация

При *инициализации* комбинируется присваивание и объявление. Например, следующий оператор объявляет переменную n_int и устанавливает ее в наибольшее допустимое значение int:

```
int n_int = INT_MAX;
```

Для инициализации значений можно использовать и литеральные константы, такие как 255. Можно инициализировать переменную другой переменной, при условии, что эта другая переменная уже была объявлена. Можно даже инициализировать переменную выражением, при условии, что все значения в выражении известны на момент объявления:

```
int uncles = 5;                    // инициализация uncles значением 5
int aunts = uncles;                // инициализация aunts значением 5
int chairs = aunts + uncles + 4;   // инициализация chairs значением 14
```

Если объявление переменной uncles переместить в конец этого списка операторов, то две других инициализации окажутся недействительными, поскольку в тот момент, когда программа попытается инициализировать остальные переменные, значение uncles не будет известно.

Синтаксис инициализации, показанный в предыдущем примере, заимствован из C; в языке C++ используется еще один синтаксис инициализации, не совместимый с C:

```
int owls = 101;  // традиционная инициализация в C; owls устанавливается в 101
int wrens(432); // альтернативный синтаксис C++; wrens устанавливается в 432
```

Внимание!

Если вы не инициализируете переменную, которая определена внутри функции, то ее значение будет *неопределенным*. Это означает, что ее значением будет случайная строка, находящаяся в ячейке памяти перед созданием переменной.

Если вы знаете, каким должно быть исходное значение переменной, инициализируйте ее. Правда, при разделении объявления переменной и присваивания ей значения может возникнуть кратковременная неопределенность:

```
short year;                        // Что бы это могло быть?
year = 1492;                       // А вот что.
```

Кроме того, инициализация переменной во время объявления предотвращает ситуацию, когда позже забывают присвоить ей значение.

Инициализация в C++11

Существует другой формат для инициализации, который используется с массивами и структурами, а в C++98 может также применяться с переменными, имеющими единственное значение:

```
int hamburgers= {24};              // устанавливает hamburgers в 24
```

Использование инициализатора с фигурными скобками для переменной, имеющей единственное значение, не было повсеместной практикой, но стандарт C++11 расширяет этот способ. Во-первых, такой инициализатор можно применять с или без знака =:

```
int emus{7};                       // устанавливает emus в 7
int rheas = {12};                  // устанавливает rheas в 12
```

Во-вторых, фигурные скобки можно оставить пустыми, тогда переменная будет инициализироваться 0:

```
int rocs = {};                    // устанавливает rocs в 0
int psychics{};                   // устанавливает psychics в 0
```

В-третьих, он предоставляет лучшую защиту от ошибок преобразования типов (мы вернемся к этой теме ближе к концу этой главы).

Может возникнуть резонный вопрос: зачем в языке нужно столько альтернативных способов инициализации? Как бы странно это не выглядело, причина в том, чтобы сделать язык C++ проще для новичков.

В прошлом в C++ использовались разные формы инициализации для различных типов, и форма, применяемая для инициализации переменных класса, отличалась от формы, используемой для инициализации обычных структур — которая, в свою очередь, отличалась от формы, применяемой для простых переменных, таких как объявленные выше.

Форма инициализации с фигурными скобками была добавлена в C++ для того, чтобы сделать инициализацию обычных переменных более похожей на инициализацию переменных класса. C++11 позволяет использовать синтаксис фигурных скобок (с или без знака =) в отношении всех типов — он представляет собой универсальный синтаксис инициализации. В будущем инициализация с применением фигурных скобок может преподноситься как основная, а другие формы расцениваться как исторические недоразумения, оставленные в языке лишь для обратной совместимости.

Типы без знаков

Каждый из только что рассмотренных четырех целочисленных типов имеет беззнаковую версию, которая не может хранить отрицательные значения. За счет этого можно увеличить наибольшее значение, которое способна хранить переменная. Например, если тип `short` представляет диапазон значений от -32 768 до 32 767, то беззнаковый вариант этого типа будет представлять диапазон от 0 до 65 535.

Естественно, типы без знаков следует использовать только для тех величин, которые никогда не будут отрицательными, например, подсчет населения, количество бобов или число участников манифестации. Создать беззнаковые версии базовых целочисленных типов можно с помощью ключевого слова `unsigned`:

```
unsigned short change;            // тип short без знака
unsigned int rovert;              // тип int без знака
unsigned quarterback;             // тоже тип int без знака
unsigned long gone;               // тип long без знака
unsigned long long lang_lang;     // тип long long без знака
```

Обратите внимание, что указание просто `unsigned` является сокращением для `unsigned int`.

В листинге 3.2 приведен пример использования беззнаковых типов. В нем также показано, что может произойти, если программа попытается выйти за пределы, установленные для целочисленных типов. В этом листинге можно увидеть еще один пример применения препроцессорного оператора `#define`.

Листинг 3.2. exceed.cpp

```
// exceed.cpp – выход за пределы для некот     целочисленных типов
#include <iostream>
#define ZERO 0           // создает символ ZERO для значения 0
#include <climits>       // определяет INT_MAX как наибольшее значение int
int main()
{
   using namespace std;
   short sam = SHRT_MAX;      // инициализирует переменную максимальным значением
   unsigned short sue = sam;  // нормально, поскольку переменная sam уже определена
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl
        << "Add $1 to each account." << endl << "Now ";
   sam = sam + 1;
   sue = sue + 1;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited.\nPoor Sam!" << endl;
   sam = ZERO;
   sue = ZERO;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl;
   cout << "Take $1 from each account." << endl << "Now ";
   sam = sam - 1;
   sue = sue - 1;
   cout << "Sam has " << sam << " dollars and Sue has " << sue;
   cout << " dollars deposited." << endl << "Lucky Sue!" << endl;
   return 0;
}
```

Ниже показан вывод программы из листинга 3.2:

```
Sam has 32767 dollars and Sue has 32767 dollars deposited.
Add $1 to each account.
Now Sam has -32768 dollars and Sue has 32768 dollars deposited.
Poor Sam!
Sam has 0 dollars and Sue has 0 dollars deposited.
Take $1 from each account.
Now Sam has -1 dollars and Sue has 65535 dollars deposited.
Lucky Sue!
```

В этой программе переменным типа `short` (`sam`) и `unsigned short` (`sue`) присваивается максимальное значение `short`, которое в нашей системе составляет 32 767. Затем к значению каждой переменной прибавляется 1. С переменной `sue` все проходит гладко, поскольку новое значение меньше максимального значения для целочисленного беззнакового типа. А переменная `sam` вместо 32 767 получает значение –32 768! Аналогично и при вычитании единицы от нуля для переменной `sam` все пройдет гладко, а вот беззнаковая переменная `sue` получит значение 65 535. Как видите, эти целые числа ведут себя почти так же, как и счетчик пробега. После достижения предельного значения отсчет начнется с другого конца диапазона (рис. 3.1).

Язык C++ гарантирует, что беззнаковые типы ведут себя именно таким образом. Однако C++ не гарантирует, что для целочисленных типов со знаком можно выходить за пределы (переполнение и потеря значимости) без сообщения об ошибке; с другой стороны, это самое распространенное поведение в современных реализациях.

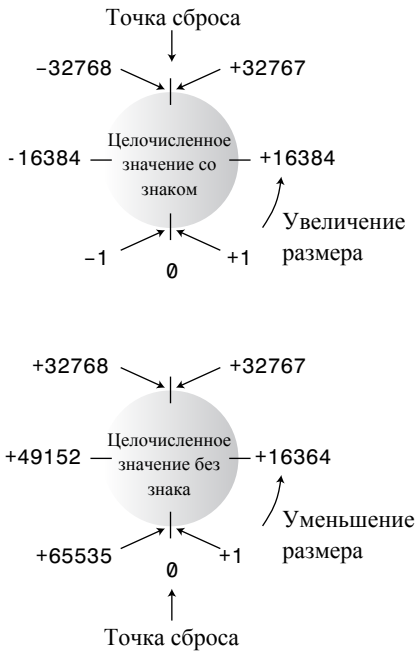


Рис. 3.1. *Типичное поведение при переполнении для целочисленных типов*

Выбор целочисленного типа

Какой из целочисленных типов следует использовать, учитывая богатство их выбора в C++? В общем случае, тип `int` имеет наиболее "естественный" размер целого числа для целевого компьютера. Под *естественным размером* подразумевается целочисленная форма, которую компьютер может обработать наиболее эффективным образом. Если нет веской причины для выбора другого типа, то лучше всего использовать тип `int`.

Теперь давайте рассмотрим причины, по которым может быть избран другой тип. Если переменная представляет какую-то величину, которая никогда не будет отрицательной, например, количество слов в документе, можно использовать тип без знака; таким образом, переменная сможет представлять более высокие значения.

Если вы знаете, что переменная будет содержать целочисленные значения, слишком большие для 16-битного целочисленного типа, применяйте тип `long`. Это справедливо даже в ситуации, когда в системе тип `int` занимает 32 бита. В результате, если вы перенесете программу в систему, которая работает с 16-битными значениями `int`, сбоев в работе программы не случится (рис. 3.2). Если же вы имеете дело с очень большими значениями, выберите тип `long long`.

Использование типа `short` позволит сократить потребление памяти, если `short` занимает меньше пространства, чем `int`. Обычно это важно только при работе с крупным массивом целых чисел. (*Массив* представляет собой структуру данных, которая хранит множество значений одного типа последовательно в памяти.) Если перед вами действительно встает вопрос экономии памяти, то вместо типа `int` предпочтительнее использовать тип `short`, даже если оба имеют одинаковый размер.

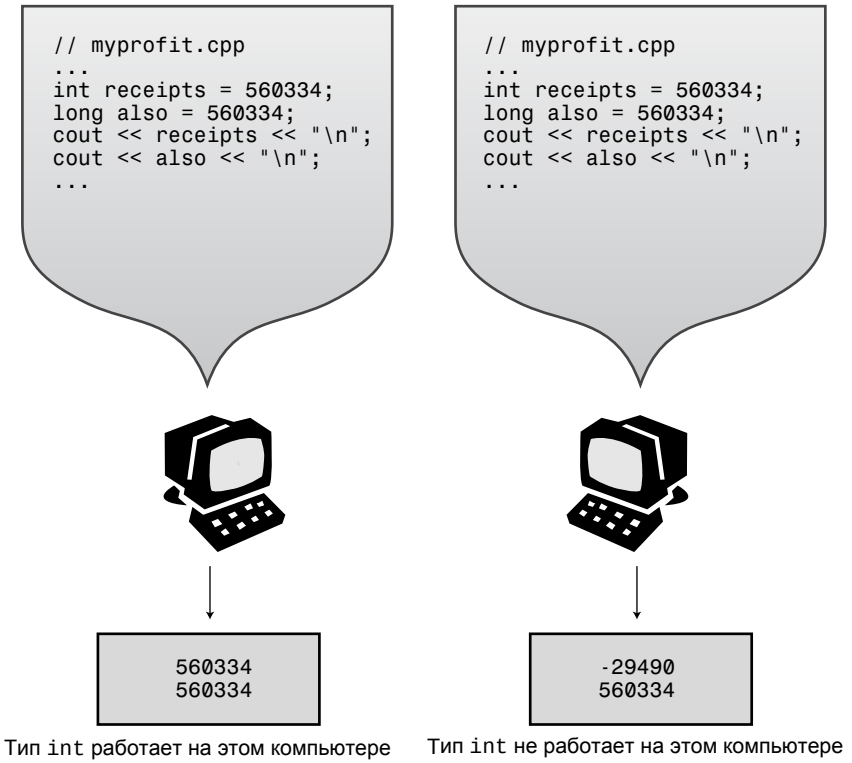


Рис. 3.2. *Чтобы обеспечить переносимость программы, используйте для больших целых чисел тип* `long`

Предположим, например, что вы переносите свою программу из системы с 16-битным `int` в систему с 32-битным `int`. В результате этого объем памяти, необходимый для хранения массива `int`, будет удвоен, но требования к массиву `short` не изменятся. Всегда помните о том, что пользу приносит каждый сэкономленный бит информации.

Если вам необходим только одиночный байт, можете применять тип `char`. Вскоре мы рассмотрим этот тип.

Целочисленные литералы

Целочисленный литерал, или константа, представляет собой число, записываемое явно, такое как 212 или 1776. Язык C++, как и C, позволяет записывать целые числа в трех различных системах счисления: с основанием 10 (наиболее распространенная форма), с основанием 8 (старая запись в системах семейства Unix) и с основанием 16 (излюбленная форма для тех, кто имеет дело с оборудованием). Описание этих систем можно найти в приложении A, а сейчас мы будем рассматривать их представления в C++. Для идентификации основания числовой константы в C++ используется первая одна или две цифры. Если первая цифра находится в диапазоне 1–9, то это число десятичное (с основанием 10); т.е. основанием 93 является 10. Если первой цифрой является 0, а вторая цифра находится в диапазоне 1–7, то это число восьмеричное (основание 8); таким образом, 042 — это восьмеричное значение, соответ-

ствующее десятичному числу 34. Если первыми двумя символами являются 0x или 0X, то это шестнадцатеричное значение (основание 16); поэтому 0x42 — это шестнадцатеричное значение, соответствующее десятичному числу 66. В представлении десятичных значений символы a–f и A–F представляют шестнадцатеричные цифры, соответствующие значениям 10–15. То есть 0xF — это 15, а 0xA5 — это 165 (10 раз по шестнадцать плюс 5). В листинге 3.3 приведен пример.

Листинг 3.3. hexoct1.cpp

```
// hexoct1.cpp -- показывает шестнадцатеричные и восьмеричные литералы
#include <iostream>
int main()
{
   using namespace std;
   int chest = 42;        // десятичный целочисленный литерал
   int waist = 0x42;      // шестнадцатеричный целочисленный литерал
   int inseam = 042;      // восьмеричный целочисленный литерал

   cout << "Monsieur cuts a striking figure!\n";
   cout << "chest = " << chest << " (42 in decimal)\n";
   cout << "waist = " << waist << " (0x42 in hex)\n";
   cout << "inseam = " << inseam << " (042 in octal)\n";
   return 0;
}
```

По умолчанию cout отображает целые числа в десятичной форме, независимо от их записи в программе, как показано в следующем выводе:

```
Monsieur cuts a striking figure!
chest = 42 (42 in decimal)
waist = 66 (0x42 in hex)
inseam = 34 (042 in octal)
```

Имейте в виду, что эти системы обозначений служат просто для удобства. Например, если вы относитесь к владельцам древних ПК и прочитали в документации, что сегментом видеопамяти CGA является B000 в шестнадцатеричном представлении, то вам не придется переводить его в значение 45 056 десятичного формата, прежде чем применять в своей программе. Вместо этого можете использовать форму 0xB000. Вне зависимости от того, как вы запишете число десять — 10, 012 или 0xA — в памяти компьютера оно будет храниться как двоичное (с основанием 2) значение.

Кстати, если нужно отобразить значение в шестнадцатеричной или восьмеричной форме, то для этого можно воспользоваться возможностями объекта cout. Вспомните, что заголовочный файл iostream предлагает манипулятор endl, который сигнализирует объекту cout о начале новой строки. Кроме этого манипулятора существуют манипуляторы dec, hex и oct, которые сообщают объекту cout формат отображения целых чисел: десятичный, шестнадцатеричный и восьмеричный, соответственно. В листинге 3.4 манипуляторы hex и oct применяются для отображения десятичного значения 42 в трех формах. (Десятичная форма используется по умолчанию, и каждая форма записи остается в силе до тех пор, пока она явно не будет изменена.)

Листинг 3.4. hexoct2.cpp

```
// hexoct2.cpp -- отображает значения в шестнадцатеричном и восьмеричном форматах
#include <iostream>
using namespace std;
```

```
int main()
{
    using namespace std;
    int chest = 42;
    int waist = 42;
    int inseam = 42;

    cout << "Monsieur cuts a striking figure!" << endl;
    cout << "chest = " << chest << " (decimal for 42)" << endl;
    cout << hex;     // манипулятор для изменения основания системы счисления
    cout << "waist = " << waist << " (hexadecimal for 42)" << endl;
    cout << oct;     // манипулятор для изменения основания системы счисления
    cout << "inseam = " << inseam << " (octal for 42)" << endl;
    return 0;
}
```

Ниже показан вывод программы из листинга 3.4:

```
Monsieur cuts a striking figure!
chest = 42 (decimal for 42)
waist = 2a (hexadecimal for 42)
inseam = 52 (octal for 42)
```

Обратите внимание, что код, подобный следующему, ничего не отображает на экране:

```
cout << hex;
```

Вместо этого он изменяет способ отображения целых чисел. Поэтому манипулятор `hex` на самом деле является сообщением для `cout`, на основании которого определяется дальнейшее его поведение. Также обратите внимание, что поскольку идентификатор `hex` является частью пространства имен `std`, которое используется в программе, применять `hex` в качестве имени переменной нельзя. Однако если опустить директиву `using`, а взамен использовать `std::cout`, `std::endl`, `std::hex` и `std::oct`, тогда `hex` можно будет выбирать для имени переменной.

Определение компилятором C++ типа константы

Объявления в программе сообщают компилятору C++ тип каждой целочисленной переменной. А что с константами? Предположим, что в программе вы представляете число с помощью константы:

```
cout << "Year = " << 1492 << "\n";
```

В каком виде программа сохраняет значение `1492`: `int`, `long` или в форме другого целочисленного типа? Ответ таков: C++ хранит целочисленные константы в виде `int`, если только нет причины поступать по-другому. Таких причин может быть две: вы используете специальный суффикс, чтобы указать конкретный тип; значение слишком большое, чтобы его можно было уместить в `int`.

Для начала посмотрим, что собой представляют суффиксы. Это буквы, размещаемые в конце числовой константы для обозначения типа константы. Суффикс `l` или `L` в целом числе означает, что оно относится к типу `long`, суффикс `u` или `U` определяет константу как `unsigned int`, а суффикс `ul` (в любой комбинации символов и в любом регистре) обозначает константу `unsigned long`. (Поскольку внешний вид буквы `l` в нижнем регистре очень похож на цифру `1`, рекомендуется использовать `L` в верхнем регистре.) Например, в системе, в которой используется 16-битный `int` и 32-битный `long`, число `22022` сохраняется в 16 битах как `int`, а число `22022L` — в 32 битах как `long`.

Подобным образом 22022LU и 22022UL получают тип unsigned long. В C++11 дополнительно предоставляются суффиксы ll и LL для типа long long, а также ull, Ull, uLL и ULL для типа unsigned long long.

Теперь давайте поговорим о размере. В C++ правила для десятичных целых чисел несколько отличаются от таковых для шестнадцатеричных и восьмеричных целых чисел. (Здесь под десятичными подразумеваются числа с основанием 10, а под шестнадцатеричными — числа с основанием 16; термин *десятичный* не обязательно подразумевает наличие десятичной точки.) Десятичное целое число без суффикса будет представлено наименьшим из следующих типов, которые могут его хранить: int, long или long long. В системе, в которой применяется 16-битный int и 32-битный long, число 20000 представляется как int, число 40000 — как long, а 3000000000 — как long long.

Шестнадцатеричное или восьмеричное целое число без суффикса будет представлено наименьшим из следующих типов, которые могут его хранить: int, long, unsigned long, long long или unsigned long long. В той же системе, в которой число 40000 представляется как long, его шестнадцатеричный эквивалент 0x9C40 получит тип unsigned int. Это объясняется тем, что шестнадцатеричная форма часто используется для выражения адресов памяти, которые по определению не могут быть отрицательными. Поэтому тип unsigned int больше подходит для 16-битных адресов, нежели long.

Тип char: символы и короткие целые числа

Пришло время рассмотреть последний целочисленный тип: char. Исходя из своего названия (сокращение от *character* — символ), char предназначен для хранения символов, таких как буквы и цифры. Хранение чисел в памяти компьютера не представляет сложности, тогда как хранение букв связано с рядом проблем. В языках программирования принят простой подход, при котором для букв используются числовые коды.

Таким образом, тип char является еще одним целочисленным типом. Для целевой компьютерной системы гарантируется, что char будет настолько большим, чтобы представлять весь диапазон базовых символов — все буквы, цифры, знаки препинания и т.п. На практике многие системы поддерживают менее 128 разновидностей символов, поэтому одиночный байт может представлять весь диапазон. Следовательно, несмотря на то, что тип char чаще всего служит для хранения символов, его можно применять как целочисленный тип, который обычно меньше, чем short.

Самым распространенным набором символов в США является ASCII, который описан в приложении B. Каждый символ в этом наборе представлен числовым кодом (кодом ASCII). Например, символу A соответствует код 65, символу M — код 77 и т.д. Для удобства в примерах этой книги предполагается использование кодов ASCII. Тем не менее, любая реализация C++ будет работать с кодом, встроенным в систему, например, EBCDIC в мэйнфреймах IBM.

Ни ASCII, ни EBCDIC не в состоянии полностью представить все интернациональные символы, поэтому в C++ поддерживается расширенный символьный тип, способный хранить более широкий диапазон значений (таких как входящие в международный набор символов Unicode). Этот тип wchar_t будет рассматриваться далее в этой главе.

Пример использования типа char приведен в листинге 3.5.

Листинг 3.5. chartype.cpp

```
// chartype.cpp -- тип char
#include <iostream>
int main( )
{
   using namespace std;
   char ch;       // объявление переменной char

   cout << "Enter a character: " << endl;
   cin >> ch;
   cout << "Hola! ";
   cout << "Thank you for the " << ch << " character." << endl;
   return 0;
}
```

Ниже показан вывод программы из листинга 3.5:

```
Enter a character:
M
Hola! Thank you for the M character.
```

Интересный момент здесь в том, что вы вводите символ M, а не соответствующий код символа 77. Кроме того, программа выводит на экран символ M, а не код 77. Вдобавок, если заглянуть в память, выяснится, что 77 — это значение, которое хранится в переменной ch. Подобного рода "магия" связана не типом char, а с объектами cin и cout, которые выполняют все необходимые преобразования. На входе объект cin преобразует нажатие клавиши <M> в значение 77. На выходе объект cout преобразует значение 77 для отображения символа M; поведение cin и cout зависит от типа переменной. Если то же значение 77 присвоить переменной типа int, объект cout отобразит для нее 77 (т.е. два символа 7). Это демонстрируется в листинге 3.6. В нем также показано, как записывается символьный литерал в C++: символ заключается в одинарные кавычки, как в 'M'. (Обратите внимание, что в этом примере не используются двойные кавычки. В C++ одинарные кавычки применяются для символов, а двойные кавычки — для строк. Объект cout умеет обрабатывать оба вида кавычек, но, как будет показано в главе 4, между ними есть существенные различия.) Наконец, в этой программе продемонстрировано еще одно средство cout — функция cout.put(), отображающая одиночный символ.

Листинг 3.6. morechar.cpp

```
// morechar.cpp -- сравнение типов char и int
#include <iostream>
int main()
{
   using namespace std;
   char ch = 'M';          // присваивает ch код ASCII символа M
   int i = ch;             // сохраняет этот же код в int
   cout << "The ASCII code for " << ch << " is " << i << endl;

   cout << "Add one to the character code:" << endl;
   ch = ch + 1;            // изменяет код символа в ch
   i = ch;                 // сохраняет код нового символа в i
   cout << "The ASCII code for " << ch << " is " << i << endl;

   // Использование функции-члена cout.put() для отображения символа
   cout << "Displaying char ch using cout.put(ch): ";
   cout.put(ch);
```

```
    // Использование cout.put() для отображения символьной константы
    cout.put('!');
    cout << endl << "Done" << endl;
    return 0;
}
```

Ниже показан вывод программы из листинга 3.6:

```
The ASCII code for M is 77
Add one to the character code:
The ASCII code for N is 78
Displaying char ch using cout.put(ch): N!
Done
```

Замечания по программе

В программе из листинга 3.6 обозначение `'M'` представляет числовой код символа `M`, поэтому инициализация переменной `ch` типа `char` значением `'M'` приводит к ее установке в 77. Затем такое же значение присваивается переменной `i` типа `int`, так что обе переменных имеют значение 77. Далее объект `cout` отображает значение `ch` как `M`, а значение `i` — как 77. Ранее уже утверждалось, что объект `cout` выбирает способ отображения значения на основе его типа — еще один пример поведения интеллектуальных объектов.

Поскольку переменная `ch` на самом деле является целочисленной, над ней можно выполнять целочисленные операции, такие как добавление 1. В результате значение `ch` изменится на 78. Это новое значение затем было присвоено переменной `i`. (Эквивалентный результат дало бы прибавление 1 к `i`.) И в данном случае объект `cout` отображает вариант `char` этого значения в виде буквы и вариант `int` в виде числа.

Факт представления символов в C++ в виде целых чисел является большим удобством, поскольку существенно облегчается манипулирование символьными значениями. Никаких функций для преобразования символов в коды ASCII и обратно использовать не придется.

Даже цифры, вводимые с клавиатуры, читаются как символы. Взгляните на следующий фрагмент:

```
char ch;
cin >> ch;
```

Если вы наберете `5` и нажмете клавишу <Enter>, этот фрагмент прочитает символ `5` и сохранит код для символа `5` (53 в ASCII) в переменной `ch`. Теперь посмотрите на такой фрагмент:

```
int n;
cin >> n;
```

Тот же самый ввод приводит к тому, что программа прочитает символ `5` и запустит подпрограмму, преобразующую символ в соответствующее числовое значение `5`, которое и сохранится в переменной `n`.

И, наконец, с помощью функции `cout.put()` в программе отображается значение переменной `ch` и символьная константа.

Функция-член: `cout.put()`

Так что же собой представляет функция `cout.put()` и почему в ее имени присутствует точка? Эта функция является самым первым примером важной концепции

ООП – это *функция-член*. Вспомните, что класс определяет способ представления данных и действия, которые можно над ними выполнять. Функция-член принадлежит к классу и описывает способ манипулирования данными класса. Класс ostream, например, имеет функцию-член put(), которая предназначена для вывода символов. Функцию-член можно применять только с определенным объектом из этого класса, таким как cout в примере. Чтобы использовать функцию-член класса с объектом, подобным cout, необходимо с помощью точки скомбинировать имя объекта (cout) и имя функции (put()). Точка в таком случае называется *операцией членства*. Нотация cout.put() означает использование функции-члена put() и объекта класса cout.

Более детально об этом речь пойдет в главе 10. А пока что единственными классами, с которыми мы имеем дело, являются istream и ostream, поэтому вы можете поэкспериментировать с их функциями-членами, чтобы лучше ознакомиться с концепцией.

Функция-член cout.put() предлагает альтернативу применению операции << для отображения символа. В этот момент вас может удивить необходимость в существовании этой функции. Причины носят в основном исторический характер. До выхода версии 2.0 языка C++ объект cout отображал символьные *переменные* в виде символов, но символьные *константы* (такие как 'M' и 'N') – в виде чисел. Проблема заключалась в том, что в ранних версиях C++, как и в C, символьные константы хранились как тип int. То есть код 77 для 'M' мог быть сохранен в 16- и 32-битном участке памяти. Между тем, переменные типа char обычно занимали 8 битов. Приведенный ниже оператор копировал 8 битов (значащие 8 битов) из константы 'M' в переменную ch:

```
char ch = 'M';
```

К сожалению, это означает, что для объекта cout константа 'M' и переменная ch выглядят совершенно разными, даже если они содержат одно и то же значение. Поэтому следующий оператор печатал код ASCII для символа $, а не символ $:

```
cout << '$';
```

Однако приведенный ниже вызов печатала требуемый символ:

```
cout.put('$');
```

Теперь, после выхода версии C++ 2.0, односимвольные константы сохраняются в виде char, а не int, поэтому объект cout правильно обрабатывает символьные константы. Объект cin читает символы из ввода двумя разными способами. Эти способы предусматривают использование циклов, поэтому их рассмотрение откладывается до главы 5.

Литералы char

Символьные литералы в C++ можно записать несколькими способами. Обычные символы, такие как буквы, знаки препинания и цифры, проще всего заключать в одиночные кавычки. Такая форма записи будет символизировать числовой код символа. Например, в системе ASCII установлены следующие соответствия:

- 'A' соответствует 65, коду ASCII для символа A;
- 'a' соответствует 97, коду ASCII для символа a;
- '5' соответствует 53, коду ASCII для цифры 5;
- ' ' соответствует 32, коду ASCII для символа пробела;
- '!' соответствует 33, коду ASCII для символа восклицательного знака.

Применять такую нотацию лучше, чем явные числовые коды, поскольку ее форма понятна и не требует запоминания конкретных кодов. Если в системе используется EBCDIC, то код 65 не будет соответствовать букве A, в то время как 'A' будет по-прежнему представлять символ.

Некоторые символы невозможно ввести в программу напрямую с клавиатуры. Например, вы не сможете сделать символ новой строки частью строки, нажав клавишу <Enter>; вместо этого редактор текста программы будет интерпретировать нажатие клавиши как запрос на начало новой строки в файле исходного кода. С использованием других символов связаны сложности, заключающиеся в том, что в языке C++ они имеют особое значение. Например, символ двойной кавычки ограничивает строковые литералы, поэтому в середине строкового литерала его вводить нельзя. Для некоторых таких символов в языке C++ применяются специальные обозначения, называемые *управляющими последовательностями* (табл. 3.2). Например, последовательность \a представляет символ предупреждения, по которому динамик издает сигнал. Последовательность \n представляет новую строку. Последовательность \" представляет двойную кавычку как обычный символ, а не ограничитель строки. Эти обозначения можно применять в строках или символьных литералах, как показано ниже:

```
char alarm = '\a';
cout << alarm << "Don't do that again!\a\n";
cout << "Ben \"Buggsie\" Hacker\nwas here!\n";
```

Таблица 3.2. Коды управляющих последовательностей в C++

Название символа	Символ ASCII	Код C++	Десятичный код ASCII	Шестнадцатеричный код ASCII
Новая строка	NL (LF)	\n	10	0xA
Горизонтальная табуляция	HT	\t	9	0x9
Вертикальная табуляция	VT	\v	11	0xB
Забой	BS	\b	8	0x8
Возврат каретки	CR	\r	13	0xD
Предупреждение	BEL	\a	7	0x7
Обратная косая черта	\	\\	92	0x5C
Знак вопроса	?	\?	63	0x3F
Одинарная кавычка	'	\'	39	0x27
Двойная кавычка	"	\"	34	0x22

Последняя строка выдает следующий результат:

```
Ben "Buggsie" Hacker
was here!
```

Обратите внимание, что управляющие последовательности, такие как \n, рассматриваются как обычные символы вроде Q. То есть она заключается в одинарные кавычки для создания символьной константы и указывается без кавычек при включении в качестве части строки.

Концепция управляющих последовательностей относится к тем временам, когда взаимодействие человека с компьютером осуществлялось с помощью телетайпа — электромеханической пишущей машинки-принтера — и современные системы

не всегда воспринимают полный набор управляющих последовательностей. Например, некоторые системы не реагируют на символ предупреждения.

Символ новой строки является альтернативной манипулятору `endl` для вставки новых строк в вывод. Его можно использовать в виде символьной константы (`'\n'`) или символа в строке (`"\n"`). Каждый из представленных далее вариантов перемещает курсор на экране на начало следующей строки:

```
cout << endl;           // использование манипулятора endl
cout << '\n';           // использование символьной константы
cout << "\n";           // использование строки
```

Символ новой строки можно включать в более длинную строку; чаще всего этот вариант является более удобным, чем использование `endl`. Например, следующие два оператора `cout` дают одинаковый вывод:

```
cout << endl << endl << "What next?" << endl << "Enter a number:" << endl;
cout << "\n\nWhat next?\nEnter a number:\n";
```

Для отображения числа легче ввести манипулятор `endl`, чем `"\n"` или `'\n'`, а для отображения строки проще использовать символ новой строки:

```
cout << x << endl;      // проще, чем cout << x << "\n";
cout << "Dr. X.\n";     // проще, чем cout << "Dr. X." << endl;
```

Наконец, можно применять управляющие последовательности на основе восьмеричных или шестнадцатеричных кодов символа. Например, комбинация клавиш <Ctrl+Z> имеет ASCII-код `26`, соответствующий восьмеричному значению `032` и шестнадцатеричному `0x1a`. Этот символ можно представить с помощью одной из управляющих последовательностей: `\032` или `\x1a`. Из них можно сделать символьные константы, заключив в одинарные кавычки, например, `'\032'`, а также использовать в виде части строки, например, `"hi\x1a there"`.

> **Совет**
>
> Когда имеется возможность выбора между числовыми и символьными управляющими последовательностями, например, между `\0x8` и `\b`, используйте символьный код. Числовое представление привязано к определенному коду, такому как ASCII, а символьное представление работает со всеми кодами и более читабельно.

В листинге 3.7 демонстрируется применение некоторых управляющих последовательностей. В нем используется символ предупреждения для привлечения внимания, символ новой строки для перемещения курсора, а также символ забоя для возврата курсора на одну позицию влево.

Листинг 3.7. `bondini.cpp`

```
// bondini.cpp -- использование управляющих последовательностей
#include <iostream>
int main()
{
   using namespace std;
   cout << "\aOperation \"HyperHype\" is now activated!\n";
   cout << "Enter your agent code:________\b\b\b\b\b\b\b\b";
   long code;
   cin >> code;
   cout << "\aYou entered " << code << "...\n";
   cout << "\aCode verified! Proceed with Plan Z3!\n";
   return 0;
}
```

На заметку!

Некоторые системы могут вести себя по-разному — отображать \b в виде небольшого прямоугольника вместо забоя, очищать предыдущую позицию либо игнорировать \a.

Если выполнить программу из листинга 3.7, то на экране появится следующий текст:

```
Operation "HyperHype" is now activated!
Enter your agent code:_____
```

После отображения символов подчеркивания программа использует символ забоя для возврата курсора на первый символ подчеркивания. После этого вы можете ввести секретный код, и выполнение программы продолжится дальше. Ниже показан полный результат ее выполнения:

```
Operation "HyperHype" is now activated!
Enter your agent code:42007007
You entered 42007007...
Code verified! Proceed with Plan Z3!
```

Универсальные символьные имена

В реализациях C++ поддерживается базовый набор символов, которые можно применять для написания исходного кода. Он состоит из букв (в верхнем и нижнем регистрах) и цифр стандартной клавиатуры США, символов, таких как { и =, используемых в языке C, и ряда других символов, включая пробельные. Существует также и базовый набор символов для выполнения, который включает символы, обрабатываемые во время выполнения программы (например, символы, прочитанные из файла или отображаемые на экране). Это добавляет несколько других символов, таких как забой и сигнал предупреждения. Стандарт C++ разрешает также реализацию для работы с расширенными наборами символов для исходного кода и выполнения. Более того, дополнительные символы, которые квалифицируются как буквы, могут использоваться как часть имени идентификатора. Так, в немецкой реализации допускается использование умляутов, а во французской — гласных со знаками ударения. В языке C++ имеется механизм представления таких интернациональных символов, которые не зависят от конкретной клавиатуры: использование *универсальных имен символов*.

Применение универсальных имен символов подобно использованию управляющих последовательностей. Универсальное имя символа начинается с последовательности \u или \U. За последовательностью \u следуют 8 шестнадцатеричных цифр, а за последовательностью \U — 16 шестнадцатеричных цифр. Эти цифры представляют код ISO 10646 для символа. (ISO 10646 является международным стандартом, разработка которого пока не завершена; он предлагает числовые коды для широкого диапазона символов. См. врезку "Unicode и ISO 10646" далее в главе.)

Если ваша реализация поддерживает расширенный набор символов, то универсальные имена символов можно применять в идентификаторах, в символьных константах и в строках. Взгляните на следующий фрагмент кода:

```
int k\u00F6rper;
cout << "Let them eat g\u00E2teau.\n";
```

Кодом ISO 10646 для символа ö является 00F6, а для символа â — 00E2. Поэтому в приведенном фрагменте переменной будет присвоено имя körper и отображена следующая строка:

```
Let them eat gâteau.
```

Если ваша система не поддерживает ISO 10646, она может отобразить какой-нибудь другой символ вместо â или, возможно, слово gu00E2teau.

На самом деле с точки зрения читабельности нет особого смысла использовать \u00F6 в качестве части имени переменной, но реализация, включающая символ ö в набор символов для исходного кода, возможно, также позволит вводить этот символ с клавиатуры.

Обратите внимание, что в C++ применяется термин "универсальное кодовое имя", а не, скажем, "универсальный код". Причина в том, что конструкция наподобие \u00F6 должна рассматриваться как метка, означающая "символ с кодом Unicode, равным U-00F6". Совместимый компилятор C++ распознает это как представление символа 'ö', но не существует никаких требований, чтобы внутренним кодом был именно 00F6. Подобно тому, как в принципе символ 'T' может быть внутренне представлен схемой ASCII на одном компьютере и другой кодовой системой на другом, символ '\u00F6' может иметь разные коды в различных системах. В исходном коде может применяться одно и то же универсальное кодовое имя для всех систем, а компилятор затем будет представлять его соответствующим внутренним кодом, используемым в конкретной системе.

Unicode и ISO 10646

Unicode предлагает решение для представления различных наборов символов за счет предоставления системы счисления для большого количества символов, сгруппированных по типам. Например, кодировка ASCII является подмножеством Unicode, поэтому буквы латинского алфавита для США (такие как A и Z) имеют одинаковое представление в обеих системах. Unicode также включает другие символы латинского алфавита, которые употребляются в европейских языках, буквы из других алфавитов, включая греческий, кириллицу, иврит, чероки, арабский, тайский и бенгальский, а также иероглифы, используемые китайцами и японцами. К настоящему времени Unicode представляет более 109 000 символов и свыше 90 письменностей, и в настоящий момент работа над этим кодом продолжается. Дополнительные сведения можно получить на веб-сайте консорциума Unicode по адресу www.unicode.org.

Unicode назначает каждому своему символу число, называемое кодовой позицией. Типовое обозначение кодовой позиции Unicode выглядит как U-222B. Здесь U указывает на то, что это символ Unicode, а 222B представляет собой шестнадцатеричное число для символа — знака интеграла в данном случае.

В Международной организации по стандартизации (ISO) была сформирована рабочая группа по разработке стандарта ISO 10646, который также является стандартом для кодировки многоязычных текстов. Группы ISO 10646 и Unicode ведут совместную работу с 1991 г., синхронизируя эти стандарты друг с другом.

Типы signed char и unsigned char

В отличие от int, тип char по умолчанию не имеет знака. Кроме того, по умолчанию он не является также и беззнаковым типом. Выбор необходимого варианта оставлен за реализацией C++, что позволяет разработчикам компиляторов подбирать такой тип, который наиболее подходит для аппаратных средств. Если для вас крайне важно, чтобы тип char обладал определенным поведением, можете использовать типы signed char и unsigned char явным образом:

```
char fodo;          // может быть со знаком, а может быть и без знака
unsigned char bar;  // явное указание беззнакового типа
signed char snark;  // явное указание типа со знаком
```

Эти отличия будут особенно важными, если тип char используется в качестве числового типа. Тип unsigned char обычно представляет диапазон значений от 0 до 255, а signed char – диапазон от –128 до 127. Предположим, например, что вы хотите использовать переменную char для хранения значений больших 200. В одних системах этот вариант будет работать, а в других – нет. Однако если для этого случая использовать тип unsigned char, тогда все будет в порядке. С другой стороны, если переменную char использовать для хранения стандартных символов ASCII, то не будет никакой разницы, является char со знаком или без знака, так что можно просто выбрать char.

На случай, если требуется больше: wchar_t

Иногда программа должна обрабатывать наборы символов, которые не вписываются в одиночный байт из 8 битов (пример – система японских рукописных шрифтов (кандзи)). Язык C++ поддерживает это парой способов. Во-первых, если большой набор символов является базовым набором символов для реализации, то поставщик компилятора может определить char как 16-битовый байт или даже больше. Во-вторых, реализация может поддерживать как малый базовый набор символов, так и более крупный расширенный набор. Традиционный 8-битовый тип char может представлять базовый набор символов, а другой тип по имени wchar_t (от *wide character type* – расширенный тип символов) – расширенный набор символов. wchar_t – это целочисленный тип с объемом, достаточным для представления самого большого расширенного набора символов в системе. Этот тип имеет такой же размер и знак, как и один из остальных целочисленных типов, который называется *лежащим в основе* типом. Выбор лежащего в основе типа зависит от реализации, поэтому в одной системе это может быть unsigned short, а в другой – int.

Объекты cin и cout рассматривают ввод и вывод как потоки char, поэтому они не подходят для работы с типом wchar_t. Заголовочный файл iostream предоставляет аналогичные им объекты wcin и wcout, которые предназначены для обработки потоков wchar_t. Кроме того, можно указать, что символьная константа или строка относится к типу wchar_t, предварив ее буквой L. В следующем фрагменте кода переменной bob присваивается версия wchar_t буквы P и отображается версия wchar_t слова tall:

```
wchar_t bob = L'P';            // символьная константа типа wchar_t
wcout << L"tall" << endl;      // вывод строки типа wchar_t
```

В системе с 2-байтным типом wchar_t этот код хранит каждый символ в 2-байтном элементе памяти. В этой книге не используется тип wchar_t, однако вы должны быть осведомлены о его существовании, особенно если вам придется работать в команде разработчиков интернациональных программ либо пользоваться Unicode или ISO 10646.

Новые типы C++11: char16_t и char32_t

По мере приобретения сообществом программистов опыта с Unicode, стало очевидным, что типа wchar_t совершенно недостаточно. Оказывается, что кодировка символов и строк символов в компьютерной системе является более сложной, чем просто использование числовых значений Unicode (называемых кодовыми позициями). В частности, при кодировании строк символов полезно иметь определенный размер и поддержку знака. Однако наличие знака и размер wchar_t могут варьироваться от реализации к реализации. Поэтому в C++11 вводится тип char16_t, который является беззнаковым и занимает 16 битов, и char32_t – тоже беззнаковый, но занимаю-

щий 32 бита. В C++11 используется префикс u для символьных и строковых констант char16_t, как в u'C' и u"be good". Аналогично, для констант char32_t применяется префикс U, как в U'R' и U"dirty rat". Тип char16_t естественным образом соответствует универсальным именам символов в форме /u00F6, а тип char32_t — универсальным именам символов в форме /U0000222B. Префиксы u и U используются для указания принадлежности символьных литералов к типам char16_t и char32_t следующим образом:

```
char16_t ch1 = u'q';          // базовый символ в 16-битной форме
char32_t ch2 = U'/U0000222B'; // универсальное имя символа в 32-битной форме
```

Подобно wchar_t, типы char16_t и char32_t имеют лежащий в основе тип, которым является один из строенных целочисленных типов. Однако лежащий в основе тип в разных системах может отличаться.

Тип bool

В стандарт ISO C++ был добавлен новый тип (т.е. новый для языка C++) по имени bool. Он назван так в честь английского математика Джорджа Буля (George Boole), разработавшего математическое представление законов логики. Во время вычислений *булевская переменная* может принимать два значения: true (истина) или false (ложь). Ранее в языке C++, как и в C, булевский тип отсутствовал. Вместо этого, как будет показано в главах 5 и 6, C++ интерпретировал ненулевые значения как true, а нулевые — как false. Теперь для представления истинного и ложного значений можно использовать тип bool, а предварительно определенные литералы true и false позволяют представлять эти значения. Это значит, что можно записывать операторы, подобные следующему:

```
bool is_ready = true;
```

Литералы true и false могут быть преобразованы в тип int, причем true преобразовывается в 1, а false в 0:

```
int ans = true;           // ans присваивается 1
int promise = false;      // promise присваивается 0
```

Кроме того, любое числовое значение или значение указателя может быть преобразовано неявно (т.е. без явного приведения типов) в значение bool. Любое ненулевое значение преобразуется в true, а нулевое значение — в false:

```
bool start = -100;        // start присваивается true
bool stop = 0;            // stop присваивается false
```

После того как мы рассмотрим операторы if (в главе 6), тип bool будет довольно часто встречаться в примерах.

Квалификатор const

Теперь давайте вернемся к теме символических имен для констант. По символическому имени можно судить о том, что представляет константа. Кроме того, если константа используется во множестве мест программы и нужно изменить ее значение, то для этого достаточно будет модифицировать единственное определение. В примечании к операторам #define (см. врезку "Символические константы как средство препроцессора" ранее в этой главе) было сказано, что в языке C++ имеется более удобный способ поддержки символьных констант. Этот способ предусматривает использование ключевого слова const для изменения объявления и инициализации

переменной. Предположим, например, что требуется символическая константа для выражения количества месяцев в году. Просто введите в своей программе следующую строку:

```
const int Months = 12;    // Months — это символическая константа для 12
```

Теперь Months можно применять в программе вместо числа 12. (Просто число 12 может представлять количество дюймов в футе или количество пончиков в дюжине, тогда как имя Months явно отражает то, что представлено значением 12.) После инициализации константы, такой как Months, ее значение установлено. Компилятор не позволяет в последующем изменять значение Months. Если вы попробуете это сделать, g++ выдаст сообщение об ошибке, гласящее, что в программе встретилось присваивание переменной, предназначенной только для чтения. Ключевое слово const называется *квалификатором*, т.к. оно уточняет, что означает объявление.

Общепринятая практика предусматривает применение заглавной буквы в начале имени, чтобы Months можно было легко идентифицировать как константу. Это соглашение не является универсальным, однако при чтении кода программы оно помогает различать константы и переменные. В соответствии с другим соглашением все буквы имени приводятся в верхнем регистре; это соглашение обычно применяется к константам, созданным с помощью #define. Согласно еще одному соглашению перед именем константы ставится буква k, как в kmonths. Существует множество других соглашений. Во многих организациях приняты свои соглашения относительно кодирования, которым следуют все программисты.

Общая форма для создания константы выглядит следующим образом:

```
const тип имя = значение;
```

Обратите внимание, что константа инициализируется в объявлении. Показанный ниже код не годится:

```
const int toes;          // значение toes в этот момент не определено
toes = 10;               // слишком поздно!
```

Если не предоставить значение во время объявления константы, она получит неопределенное значение, которое не удастся изменить.

Если у вас имеется опыт в использовании C, вам может показаться, что оператор #define вполне адекватно решает эту задачу. Однако квалификатор const лучше. Во-первых, он позволяет явно указывать тип. Во-вторых, с помощью правил видимости можно ограничивать определение конкретными функциями или файлами. (Правила видимости описывают, в каких модулях известно то или иное имя; более детально это будет рассматриваться в главе 9.) В-третьих, квалификатор const можно использовать и для более сложных типов, таких как массивы и структуры, о чем речь пойдет в главе 4.

Совет

Если вы перешли на C++, имея до этого дело с языком C, и для определения символьных констант намерены пользоваться #define, лучше отдайте предпочтение const.

В ANSI C также используется квалификатор const, позаимствованный из C++. Если вы знакомы с версией const в ANSI C, то должны знать, что его версия в C++ немного отличается. Одно из отличий связано с правилами видимости; это объясняется в главе 9. Другое отличие в том, что в C++ (но не в C) значение const можно применять для объявления размеров массива. Примеры будут представлены в главе 4.

Числа с плавающей точкой

После представления всех целочисленных типов C++ можно переходить к рассмотрению типов с плавающей точкой, которые образуют вторую основную группу фундаментальных типов C++. Эти типы позволяют представлять числа с дробными частями, например, пробег на единицу расхода горючего танка M1 (0,56 мили на галлон). Также они поддерживают намного более широкий диапазон значений. Если число слишком большое, чтобы его можно было представить с помощью типа `long`, например, количество бактериальных клеток в теле человека (свыше 100 000 000 000), можно воспользоваться одним из типов с плавающей точкой.

Типы с плавающей точкой позволяют представить такие числа, как `2.5`, `3.14159` и `122442.32` — т.е. числа с дробными частями. Такие значения хранятся в памяти компьютера в виде двух частей. Одна часть представляет значение, а другая — масштабный коэффициент, который увеличивает или уменьшает значение. Рассмотрим следующую аналогию. Предположим, что есть два числа: `34.1245` и `34124.5`. Они идентичны друг другу за исключением масштабного коэффициента. Первое значение можно представить как `0.341245` (базовое значение) и 100 (масштабный коэффициент). Второе значение можно представить как `0.341245` (такое же базовое значение) и 100 000 (больший масштабный коэффициент).

Масштабный коэффициент предназначен для перемещения десятичной точки, откуда и пошел термин *плавающая точка*. В языке C++ используется аналогичный метод для внутреннего представления чисел с плавающей точкой, за исключением того, что он основан на двоичных числах, поэтому масштабирование производится с помощью множителя 2, а не 10. К счастью, вам не нужно досконально разбираться в механизме внутреннего представления чисел. Вы должны запомнить только то, что с помощью чисел с плавающей точкой можно представлять дробные очень большие и очень малые значения; при этом их внутреннее представление существенно отличается от представления целых чисел.

Запись чисел с плавающей точкой

В C++ поддерживаются два способа записи чисел с плавающей точкой. Первый из них заключается в использовании стандартной нотации, применяемой в повседневной жизни:

```
12.34          // число с плавающей точкой
939001.32      // число с плавающей точкой
0.00023        // число с плавающей точкой
8.0            // тоже число с плавающей точкой
```

Даже если дробная часть равна 0, как в `8.0`, наличие десятичной точки гарантирует, что число представлено в формате с плавающей точкой и не является целочисленным. (Стандарт C++ позволяет реализациям учитывать различные локали, например, использовать для представления десятичной точки запятую. Однако это влияет только на внешний вид чисел при вводе и выводе, но не на их представление в коде.)

Другой способ представления значений с плавающей точкой называется экспоненциальной записью и имеет вид, подобный следующему: `3.45E6`. Эта запись означает, что значение `3.45` умножается на 1 000 000; конструкция `E6` означает 10 в степени 6, т.е. 1 000 000. Поэтому `3.45E6` соответствует значению 3 450 000. В данном случае 6 называется *порядком*, а 3.45 — *мантиссой*. Ниже показаны дополнительные примеры таких записей:

```
2.52e+8         // можно использовать E или e; знак + необязателен
8.33E-4         // порядок может быть отрицательным
7E5             // то же, что и 7.0E+05
-18.32e13       // перед записью может стоять знак + или -
1.69e12         // внешний долг Бразилии в реалах
5.98E24         // масса Земли в килограммах
9.11e-31        // масса электрона в килограммах
```

Как видите, экспоненциальная запись весьма удобна для представления очень больших и очень малых чисел.

Экспоненциальная запись гарантирует, что число будет храниться в формате с плавающей точкой, даже при отсутствии десятичной точки. Обратите внимание, что можно использовать `E` либо `e`, а порядок может быть как положительным, так и отрицательным (рис. 3.3). Однако пробелы в записи не разрешены, поэтому, например, вариант `7.2 E6` является недопустимым.

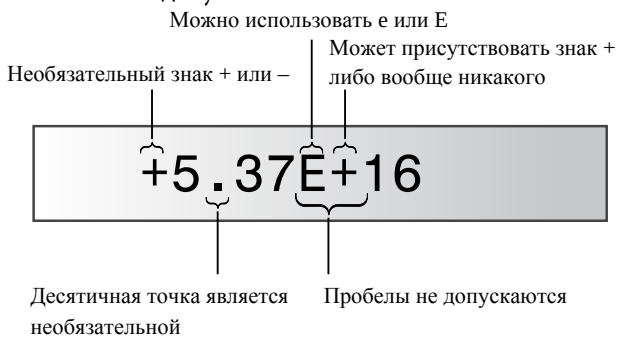


***Рис 3.3.** Экспоненциальная запись*

Отрицательное значение порядка означает деление, а не умножение на степень 10. Таким образом, `8.33E-4` означает $8.33 / 10^4$, или `0.000833`.

Точно так же рассчитывается и масса электрона в килограммах: `9.11e-31` кг равно `0.000000000000000000000000000000911`. Выбирайте тот вариант, который вам больше нравится. Обратите внимание, что `-8.33E4` означает `-83300`. Знак перед записью относится к числу, а знак в порядке применятся к масштабному коэффициенту.

На заметку!

Форма `d.dddE+n` означает перемещение десятичной точки на `n` позиций вправо, а форма `d.dddE-n` — перемещение десятичной точки на `n` позиций влево.

Типы чисел с плавающей точкой

Подобно ANCI C, язык C++ поддерживает три типа чисел с плавающей точкой: `float`, `double` и `long double`. Эти типы характеризуются количеством значащих цифр, которые они могут представлять, и минимальным допустимым диапазоном порядка. *Значащими цифрами* являются значащие разряды числа. Например, запись высоты горы Шаста (Shasta) в Калифорнии, 14 179 футов, содержит пять значащих цифр, которые определяют высоту с точностью до фута. Но запись высоты этой же горы в виде 14 000 футов использует две значащие цифры, т.е. результат округляется до ближайшей тысячи футов; в данном случае остальные три разряда являются просто заполнителями. Количество значащих цифр не зависит от позиции десятичной точки. Например, высоту можно записать как `14.179` тысяч футов. В этом случае также используются пять значащих разрядов.

Требования в C и C++ относительно количества значащих разрядов следующие: тип `float` должен иметь, как минимум, 32 бита, `double` — как минимум, 48 битов и естественно быть не меньше чем `float`, а `long double` должен быть минимум таким же, как и тип `double`. Все три типа могут иметь одинаковый размер. Однако обычно `float` занимает 32 бита, `double` — 64 бита, а `long double` — 80, 96 или 128 битов. Кроме того, диапазон порядка для каждого из этих трех типов — как минимум, от -37 до +37. Ограничения для конкретной системы можно узнать, заглянув в файл `cfloat` или `float.h`. (Файл `cfloat` является аналогом файла `float.h` в C.) Ниже в качестве примера приведены некоторые строки из файла `float.h` для Borland C++Builder:

```
// Минимальное количество значащих цифр
#define DBL_DIG 15        // double
#define FLT_DIG 6         // float
#define LDBL_DIG 18       // long double

// Количество битов, используемых для представления мантиссы
#define DBL_MANT_DIG    53
#define FLT_MANT_DIG    24
#define LDBL_MANT_DIG   64

// Максимальные и минимальные значения порядка
#define DBL_MAX_10_EXP  +308
#define FLT_MAX_10_EXP  +38
#define LDBL_MAX_10_EXP +4932

#define DBL_MIN_10_EXP  -307
#define FLT_MIN_10_EXP  -37
#define LDBL_MIN_10_EXP -4931
```

В листинге 3.8 показан пример использования типов `float` и `double` и продемонстрированы также их различия в точности, с которой они представляют числа (т.е. количество значащих цифр). В программе используется метод `ostream` по имени `setf()`, который будет рассматриваться в главе 17. В данном примере вызов этого метода устанавливает формат вывода с фиксированной точкой, который позволяет визуально определять точность. Это предотвратит переключение на экспоненциальную запись для больших чисел и заставит отображать шесть цифр справа от десятичной точки. Аргументы `ios_base::fixed` и `ios_base::floatfield` — это константы, которые доступны в результате включения `iostream`.

Листинг 3.8. `floatnum.cpp`

```
// floatnum.cpp -- типы с плавающей точкой
#include <iostream>
int main()
{
   using namespace std;
   cout.setf(ios_base::fixed, ios_base::floatfield);  // фиксированная точка
   float tub = 10.0 / 3.0;                            // подходит для 6 разрядов
   double mint = 10.0 / 3.0;                          // подходит для 15 разрядов
   const float million = 1.0e6;
   cout << "tub = " << tub;
   cout << ", a million tubs = " << million * tub;
   cout << ",\nand ten million tubs = ";
   cout << 10 * million * tub << endl;
   cout << "mint = " << mint << " and a million mints = ";
   cout << million * mint << endl;
   return 0;
}
```

Ниже показан вывод программы из листинга 3.8:

```
tub = 3.333333, a million tubs = 3333333.250000,
and ten million tubs = 33333332.000000
mint = 3.333333 and a million mints = 3333333.333333
```

Замечания по программе

Обычно объект `cout` отбрасывает завершающие нули. Например, он может отобразить `3333333.250000` как `3333333.25`. Вызов функции `cout.setf()` переопределяет это поведение, во всяком случае, в новых реализациях. Главное, на что следует обратить внимание в листинге 3.8 — тип `float` имеет меньшую точность, чем тип `double`. Переменные `tub` и `mint` инициализируются выражением `10.0 / 3.0`, результат которого равен `3.3333333333333333...` (3 в периоде). Поскольку `cout` выводит шесть цифр справа от десятичной точки, вы можете видеть, что значения `tub` и `mint` отображаются с довольно высокой точностью. Однако после того как в программе каждое число умножается на миллион, вы увидите, что значение `tub` отличается от правильного результата после седьмой тройки. `tub` дает хороший результат до седьмой значащей цифры. (Эта система гарантирует 6 значащих цифр для `float`, но это самый худший сценарий.) Переменная типа `double` показывает 13 троек, поэтому она дает хороший результат до 13-й значащей цифры. Поскольку система гарантирует 15 значащих цифр, то такой и должна быть точность этого типа. Обратите также внимание, что умножение произведения `million` и `tub` на 10 дает не совсем точный результат; это еще раз указывает на ограничения по точности типа `float`.

Класс `ostream`, к которому принадлежит объект `cout`, имеет функции-члены, которые позволяют управлять способом форматирования вывода — вы можете задавать ширину полей, количество позиций справа от десятичной точки, выбирать десятичную или экспоненциальную форму представления и т.д. Эти вопросы будут рассматриваться в главе 17. Примеры из этой книги довольно простые и обычно используют только операцию `<<`. Иногда после ее выполнения отображается больше разрядов, чем это необходимо, однако этот "дефект" всего лишь визуальный. О вариантах форматирования выходных данных вы узнаете в главе 17.

Изучение включаемых файлов

К директивам `include`, находящимся в начале файлов исходного кода C++, часто относятся как к каким-то магическим элементам; изучая материал данной книги и выполняя предложенные примеры, новички в C++ смогут узнать, для каких целей предназначены определенные заголовочные файлы, и самостоятельно их использовать для обеспечения работоспособности программ. Не воспринимайте эти файлы как нечто сверхъестественное — смелее открывайте их и внимательно изучайте. Они представляют собой обыкновенные текстовые файлы, которые можно без труда читать. Каждый файл, который вы включаете в свою программу, хранится в вашем компьютере или в доступном ему месте. Отыщите включаемые файлы и просмотрите их содержимое. Вы поймете, что используемые файлы исходного кода и заголовочные файлы являются превосходным источником информации, а в некоторых случаях в них можно найти больше сведений, чем в самой совершенной документации. Впоследствии, когда вы научитесь добавлять более сложные включения и работать с другими, нестандартными библиотеками, этот опыт принесет немалую пользу.

Константы с плавающей точкой

Когда в программе записывается константа с плавающей точкой, то с каким именно типом она будет сохранена? По умолчанию константы с плавающей точкой, такие как `8.24` и `2.4E8`, имеют тип `double`. Если константа должна иметь тип `float`, необ-

ходимо указать суффикс f или F. Для типа long double используется суффикс l или L. (Поскольку начертание буквы l в нижнем регистре очень похоже на начертание цифры 1, рекомендуется применять L в верхнем регистре.) Ниже приведены примеры использования суффиксов:

```
1.234f              // константа float
2.45E20F            // константа float
2.345324E28         // константа double
2.2L                // константа long double
```

Преимущества и недостатки чисел с плавающей точкой

Числа с плавающей точкой обладают двумя преимуществами по сравнению с целыми числами. Во-первых, они могут представлять значения, расположенные между целыми числами. Во-вторых, поскольку у них есть масштабный коэффициент, они могут представлять более широкий диапазон значений. С другой стороны, операции с плавающей точкой обычно выполняются медленнее, чем целочисленные операции, и могут приводить к потере точности (см. листинг 3.9).

Листинг 3.9. fltadd.cpp

```
// fltadd.cpp -- потеря точности при работе с float
#include <iostream>
int main()
{
   using namespace std;
   float a = 2.34E+22f;
   float b = a + 1.0f;
   cout << "a = " << a << endl;
   cout << "b - a = " << b - a << endl;
   return 0;
}
```

В программе из листинга 3.9 берется число, к нему прибавляется 1 и затем вычитается исходное число. По идее, результат должен быть равен единице. Так ли это? Ниже показан вывод программы из листинга 3.9:

```
a = 2.34e+022
b - a = 0
```

Проблема в том, что 2.34E+22 представляет число с 23 цифрами слева от десятичной точки. За счет прибавления 1 происходит попытка добавления 1 к 23-й цифре этого числа. А тип float может представлять только первые 6 или 7 цифр числа, поэтому попытка изменить 23-ю цифру не оказывает никакого воздействия на значение.

Классификация типов данных

Язык C++ привносит некоторый порядок в свои базовые типы, классифицируя их в семейства. Типы signed char, short, int и long называются *целочисленными типами со знаком*. В C++11 к этому списку добавляется тип long long. Версии без знака называются *целочисленными типами без знака* (или *беззнаковыми типами*). Типы bool, char, wchar_t, целочисленные со знаком и целочисленные без знака вместе называются *целыми* или *целочисленными* типами. В C++11 к этому списку добавляются типы char16_t и char32_t. Типы float, double и long double называются *типами с плавающей точкой*. Целочисленные типы и типы с плавающей точкой вместе называются *арифметическими* типами.

Арифметические операции в C++

Со времен средней школы у вас наверняка сохранились кое-какие воспоминания об арифметике. Теперь есть возможность переложить эту заботу на плечи компьютера. В C++ предоставляются операции для выполнения пяти базовых арифметических действий: сложения, вычитания, умножения, деления и получения остатка от деления. Каждая из этих операций использует два значения (называемые *операндами*) для вычисления конечного результата. Операция и ее операнды вместе образуют *выражение*. Например, рассмотрим следующий оператор:

```
int wheels = 4 + 2;
```

Значения `4` и `2` — это операнды, знак `+` обозначает операцию сложения, а `4 + 2` — это выражение, результатом которого является `6`.

Ниже перечислены пять базовых арифметических операций в C++.

- Операция `+` выполняет сложение операндов. Например, `4 + 20` дает `24`.
- Операция `-` вычитает второй операнд из первого. Например, `12 - 3` дает `9`.
- Операция `*` умножает операнды. Например, `28 * 4` дает `112`.
- Операция `/` выполняет деление первого операнда на второй.

  Например, `1000 / 5` дает `200`. Если оба операнда являются целыми числами, то результат будет равен целой доли частного. Например, `17 / 3` дает `5`, с отброшенной дробной частью.
- Операция `%` находит остаток от деления первого операнда на второй. Например, `19 % 6` равно `1`, поскольку `6` входит в `19` три раза, с остатком `1`. Оба операнда при этом должны быть целочисленными; использование операции `%` над числами в формате с плавающей точкой приведет к ошибке времени компиляции. Если один из операндов будет отрицательным, то знак результата удовлетворяет следующему правилу: `(a/b) *b + a%b` равно `a`.

Естественно, в качестве операндов можно использовать как переменные, так и константы. В листинге 3.10 приведен пример. Поскольку операция `%` работает только над целыми числами, мы рассмотрим ее чуть позже.

Листинг 3.10. `arith.cpp`

```
// arith.cpp -- примеры некоторых арифметических операций в C++
#include <iostream>
int main()
{
    using namespace std;
    float hats, heads;

    cout.setf(ios_base::fixed, ios_base::floatfield); // формат с фиксированной точкой
    cout << "Enter a number: ";
    cin >> hats;
    cout << "Enter another number: ";
    cin >> heads;
    cout << "hats = " << hats << "; heads = " << heads << endl;
    cout << "hats + heads = " << hats + heads << endl;
    cout << "hats - heads = " << hats - heads << endl;
    cout << "hats * heads = " << hats * heads << endl;
    cout << "hats / heads = " << hats / heads << endl;
    return 0;
}
```

Как видно в выводе программы из листинга 3.10, языку C++ можно смело поручать выполнение элементарных арифметических операций:

```
Enter a number: 50.25
Enter another number: 11.17
hats = 50.250000; heads = 11.170000
hats + heads = 61.419998
hats - heads = 39.080002
hats * heads = 561.292480
hats / heads = 4.498657
```

И все же полностью доверять ему нельзя. Так, при добавлении `11.17` к `50.25` должно получиться `61.42`, а в выводе присутствует `61.419998`. Такое расхождение не связано с ошибками выполнения арифметических операций, а объясняется ограниченными возможностями представления значащих цифр типа `float`. Если помните, для `float` гарантируется только шесть значащих цифр. Если `61.419998` округлить до шести цифр, получим `61.4200`, что является вполне корректным значением для гарантированной точности. Из этого следует, что когда нужна более высокая точность, необходимо использовать тип `double` или `long double`.

Порядок выполнения операций: приоритеты операций и ассоциативность

Можете ли вы доверить C++ выполнение сложных вычислений? Да, но для этого необходимо знать правила, которыми руководствуется C++. Например, во многих выражениях присутствует более одной операции. Тогда возникает логичный вопрос: какая из них должна быть выполнена первой? Например, рассмотрим следующий оператор:

```
int flyingpigs = 3 + 4 * 5;   // каким будет результат: 35 или 23?
```

Получается, что операнд `4` может участвовать и в сложении, и в умножении. Когда над одним и тем же операндом может быть выполнено несколько операций, C++ руководствуется правилами *старшинства* или *приоритетов*, чтобы определить, какая операция должна быть выполнена первой. Арифметические операции выполняются в соответствии с алгебраическими правилами, согласно которым умножение, деление и нахождение остатка от деления выполняются раньше операций сложения и вычитания. Поэтому выражение `3 + 4 * 5` следует читать как `3 + (4 * 5)`, но не `(3 + 4) * 5`. Таким образом, результатом этого выражения будет `23`, а не `35`. Конечно, чтобы обозначить свои приоритеты, вы можете заключать операнды и операции в скобки. В приложении Г представлены приоритеты всех операций в C++. Обратите внимание, что в приложении Г операции `*`, `/` и `%` занимают одну строку. Это означает, что они имеют одинаковый уровень приоритета. Операции сложения и вычитания имеют самый низкий уровень приоритета.

Однако знания только лишь приоритетов операций не всегда бывает достаточно. Взгляните на следующий оператор:

```
float logs = 120 / 4 * 5;     // каким будет результат: 150 или 6?
```

И в этом случае над операндом `4` могут быть выполнены две операции. Однако операции `/` и `*` имеют одинаковый уровень приоритета, поэтому программа нуждается в уточняющих правилах, чтобы определить, нужно сначала `120` разделить на `4` либо `4` умножить на `5`. Поскольку результатом первого варианта является `150`, а второго — `6`, выбор здесь очень важен. В том случае, когда две операции имеют одинаковый уровень приоритета, C++ анализирует их *ассоциативность*: слева направо или справа налево. Ассоциативность слева направо означает, что если две операции, выполняемые

над одним и тем же операндом, имеют одинаковый приоритет, то сначала выполняется операция слева от операнда. В случае ассоциативности справа налево первой будет выполнена операция справа от операнда. Сведения об ассоциативности также можно найти в приложении Г. В этом приложении показано, что для операций умножения и деления характерна ассоциативность слева направо. Это означает, что над операндом `4` первой будет выполнена операция слева. То есть `120` делится на `4`, в результате получается `30`, а затем этот результат умножается на `5`, что в итоге дает `150`.

Следует отметить, что приоритет выполнения и правила ассоциативности действуют только тогда, когда две операции относятся к одному и тому же операнду. Рассмотрим следующее выражение:

```
int dues = 20 * 5 + 24 * 6;
```

В соответствии с приоритетом выполнения операций, программа должна умножить `20` на `5`, затем умножить `24` на `6`, после чего выполнить сложение. Однако ни уровень приоритета выполнения, ни ассоциативность не помогут определить, какая из операций умножения должна быть выполнена в первую очередь. Можно было бы предположить, что в соответствии со свойством ассоциативности должна быть выполнена операция слева, однако в данном случае две операции умножения не относятся к одному и тому же операнду, поэтому эти правила здесь не могут быть применены. В действительности выбор порядка выполнения операций, который будет приемлемым для системы, оставлен за конкретной реализацией C++. В данном случае при любом порядке выполнения будет получен один и тот же результат, однако иногда результат выражения зависит от порядка выполнения операций. Мы вернемся к этому вопросу в главе 5, когда будем рассматривать операцию инкремента.

Различные результаты, получаемые после деления

Давайте продолжим рассмотрение особенностей операции деления (`/`). Поведение этой операции зависит от типа операндов. Если оба операнда являются целочисленными, то C++ выполнит целочисленное деление. Это означает, что любая дробная часть результата будет отброшена, приводя результат к целому числу. Если один или оба операнда являются значениями с плавающей точкой, то дробная часть остается, поэтому результатом будет число с плавающей точкой. В листинге 3.11 показано, как операция деления в C++ осуществляется над значениями различных типов. Как и в листинге 3.10, в листинге 3.11 вызывается функция-член `setf()` для изменения формата отображаемых результатов.

Листинг 3.11. `divide.cpp`

```
// divide.cpp -- деление целых чисел и чисел с плавающей точкой
#include <iostream>
int main()
{
    using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << "Integer division: 9/5 = " << 9 / 5 << endl;
    cout << "Floating-point division: 9.0/5.0 = ";
    cout << 9.0 / 5.0 << endl;
    cout << "Mixed division: 9.0/5 = " << 9.0 / 5 << endl;
    cout << "double constants: 1e7/9.0 = ";
    cout << 1.e7 / 9.0 << endl;
    cout << "float constants: 1e7f/9.0f = ";
    cout << 1.e7f / 9.0f << endl;
    return 0;
}
```

Ниже показан вывод программы из листинга 3.11:

```
Integer division: 9/5 = 1
Floating-point division: 9.0/5.0 = 1.800000
Mixed division: 9.0/5 = 1.800000
double constants: 1e7/9.0 = 1111111.111111
float constants: 1e7f/9.0f = 1111111.125000
```

Первая строка вывода показывает, что в результате деления целого числа `9` на целое число `5` было получено целое число `1`. Дробная часть от деления `4/5` (или `0.8`) отбрасывается. (Далее в этой главе вы увидите практический пример использования такого деления при рассмотрении операции нахождения остатка от деления.) Следующие две строки показывают, что если хотя бы один из операндов имеет формат с плавающей точкой, искомый результат (`1.8`) также будет представлен в этом формате. В действительности, при комбинировании смешанных типов C++ преобразует их к одному типу. Об этих автоматических преобразованиях речь пойдет далее в этой главе. Относительная точность в двух последних строках вывода показывает, что результат имеет тип `double`, если оба операнда имеют тип `double`, и тип `float`, если оба операнда имеют тип `float`. Не забывайте, что константы в формате с плавающей точкой имеют тип `double` по умолчанию.

Беглый взгляд на перегрузку операций

В листинге 3.11 операция деления представляет три различных действия: деление `int`, деление `float` и деление `double`. Для определения, какая операция имеется в виду, в C++ используется контекст — в рассматриваемом случае тип операндов. Использование одного и того же символа для обозначения нескольких операций называется *перегрузкой операций*. В языке C++ можно найти несколько примеров перегрузки. C++ позволяет расширить перегрузку операций на определяемые пользователем классы, поэтому то, что можно увидеть в этом примере, является предвестником важного свойства ООП (рис. 3.4).

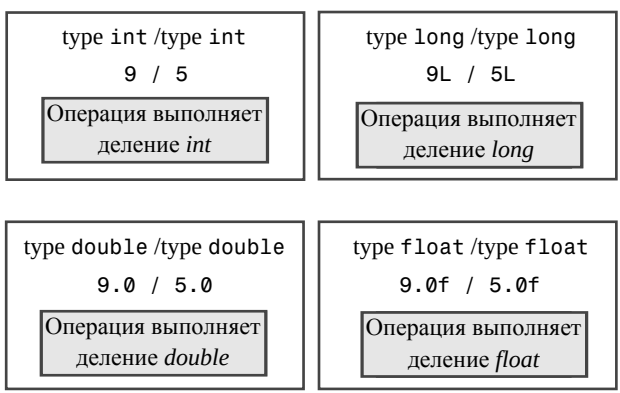


Рис. 3.4. *Различные операции деления*

Операция нахождения остатка от деления

Большинству больше знакомы операции сложения, вычитания, умножения и деления, нежели операция нахождения остатка от деления, поэтому давайте рассмотрим ее более подробно. В результате выполнения этой операции возвращается остаток, полученный от целочисленного деления. В комбинации с целочисленным делением операция нахождения остатка особенно полезна при решении задач, в которых ин-

тересуемую величину необходимо разделить на целые единицы, например, при преобразовании дюймов в футы и дюймы или долларов в двадцатипятицентовые, десятицентовые, пятицентовые и одноцентовые эквиваленты. В листинге 2.6 во второй главе был представлен пример преобразования мер веса: британских стоунов в фунты. В листинге 3.12 демонстрируется обратный процесс: преобразование фунтов в стоуны. Как вы, возможно, помните, 1 стоун равен 14 фунтам. Программа использует целочисленное деление для получения наибольшего количества целых стоунов и операцию нахождения остатка для определения оставшихся фунтов.

Листинг 3.12. modulus.cpp

```
// modulus.cpp -- использует операцию % для преобразования фунтов в стоуны
#include <iostream>
int main()
{
   using namespace std;
   const int Lbs_per_stn = 14;
   int lbs;

   cout << "Enter your weight in pounds: ";
   cin >> lbs;
   int stone = lbs / Lbs_per_stn;       // количество целых стоунов
   int pounds = lbs % Lbs_per_stn;      // остаток в фунтах
   cout << lbs << " pounds are " << stone
        << " stone, " << pounds << " pound(s).\n";
   return 0;
}
```

Ниже показан результат выполнения программы из листинга 3.12:

```
Enter your weight in pounds: 181
181 pounds are 12 stone, 13 pound(s).
```

В выражении `lbs / Lbs_per_stn` оба операнда имеют тип `int`, поэтому компьютер выполняет целочисленное деление. С учетом введенного значения (181), которое было присвоено переменной `lbs`, результат выражения оказался равным 12. Умножение 12 на 14 дает 168, поэтому остаток от деления 181 на 14 равен 13, и это значение является результатом `lbs % Lbs_per_stn`.

Преобразования типов

Большое разнообразие типов в C++ позволяет выбрать тип, удовлетворяющий необходимым требованиям. Однако помимо пользы это разнообразие еще и усложняет компьютеру жизнь. Например, при сложении двух значений, имеющих тип `short`, могут использоваться аппаратные инструкции, отличные от применяемых при сложении двух значений `long`. При наличии 11 целочисленных типов и 3 типов чисел с плавающей точкой компьютер сталкивается с множеством случаев их обработки, особенно если смешиваете различные типы. Чтобы справиться с потенциальной неразберихой в типах, многие преобразования типов в C++ осуществляются автоматически.

- C++ преобразует значения во время присваивания значения одного арифметического типа переменной, относящейся к другому арифметическому типу.
- C++ преобразует значения при комбинировании разных типов в выражениях.
- C++ преобразует значения при передаче аргументов функциям.

Если вы не понимаете, что происходит во время автоматического преобразования типов, то некоторые результаты выполнения ваших программ могут оказаться несколько неожиданными, поэтому давайте внимательно ознакомимся с правилами преобразования типов.

Преобразование при инициализации и присваивании

Язык C++ довольно либерален, разрешая присваивание числового значения одного типа переменной другого типа. Всякий раз, когда вы это делаете, значение преобразуется к типу переменной, которая его получает. Предположим, например, что переменная `so_long` имеет тип `long`, переменная `thirty` — тип `short`, а в программе присутствует следующий оператор:

```
so_long = thirty; // присваивание значения типа short переменной типа long
```

Программа принимает значение переменной `thirty` (обычно 16-битное) и расширяет его до значения `long` (обычно 32-битное) во время присваивания. Обратите внимание, что в процессе расширения создается новое значение, которое будет присвоено переменной `so_long`; содержимое переменной `thirty` остается неизменным.

Присваивание значения переменной, тип которой имеет более широкий диапазон, обычно происходит без особых проблем. Например, при присваивании значения переменной, имеющей тип `short`, переменной типа `long` само значение не изменяется; в этом случае значение просто получает несколько дополнительных байтов, которые остаются незанятыми. А если большое значение, имеющее тип `long`, например, `2111222333`, присвоить переменной типа `float`, точность будет потеряна. Поскольку переменная типа `float` может иметь только шесть значащих цифр, значение может быть округлено до `2.11122E9`. Таким образом, если одни преобразования безопасны, то другие могут привести к сложностям. В табл. 3.3 указаны некоторые возможные проблемы, связанные с преобразованиями.

Таблица 3.3. Потенциальные проблемы при преобразовании чисел

Тип преобразования	Возможные проблемы
Больший тип с плавающей точкой в меньший тип с плавающей точкой, например, `double` в `float`	Потеря точности (значащих цифр); исходное значение может превысить диапазон, допустимый для целевого типа, поэтому результат окажется неопределенным
Тип с плавающей точкой в целочисленный тип	Потеря дробной части; исходное значение может превысить диапазон целевого типа, поэтому результат будет неопределенным
Больший целочисленный тип в меньший целочисленный тип, например, `long` в `short`	Исходное значение может превысить диапазон, допустимый для целевого типа; обычно копируются только младшие байты.

Нулевое значение, присвоенное переменной `bool`, преобразуется в `false`, а ненулевое значение — в `true`.

Присваивание значений, имеющих тип с плавающей точкой, целочисленным переменным порождает пару проблем. Во-первых, преобразование значения с плавающей точкой в целочисленное приводит к усечению числа (отбрасыванию дробной части). Во-вторых, значение типа `float` может оказаться слишком большим, чтобы его уместить в переменную `int`. В этом случае C++ не определяет, каким должен быть результат, т.е. разные реализации C++ будут по-разному реагировать на подобные ситуации.

Традиционная инициализация ведет себя аналогично присваиванию. В листинге 3.13 можно увидеть несколько примеров преобразования при инициализации.

Листинг 3.13. assign.cpp

```
// assign.cpp -- изменение типа при инициализации
#include <iostream>
int main()
{
    using namespace std;
    cout.setf(ios_base::fixed, ios_base::floatfield);
    float tree = 3;             // int преобразован в float
    int guess = 3.9832;         // float преобразован в int
    int debt = 7.2E12;          // результат не определен в C++
    cout << "tree = " << tree << endl;
    cout << "guess = " << guess << endl;
    cout << "debt = " << debt << endl;
    return 0;
}
```

Ниже показан вывод программы из листинга 3.13:

```
tree = 3.000000
guess = 3
debt = 1634811904
```

В этом случае переменной `tree` присваивается значение с плавающей точкой `3.0`. Присваивание `3.9832` переменной `guess` типа `int` приводит к усечению значения до 3; при преобразовании типов с плавающей точкой в целочисленные типы в C++ значения усекаются (отбрасывается дробная часть) и не округляются (нахождение ближайшего целого числа). Наконец, обратите внимание, что переменная `debt` типа `int` не может хранить значение `7.2E12`. Это порождает ситуацию, при которой в C++ никак не определяется, каким должен быть результат. В этой системе все заканчивается тем, что `debt` получает значение `1634811904`, или примерно `1.6E09`.

Некоторые компиляторы предупреждают о возможной потере данных для тех операторов, которые инициализируют целочисленные переменные значениями с плавающей точкой. Кроме того, значение переменной `debt` варьируется от компилятора к компилятору. Например, выполнение программы из листинга 3.13 на второй системе даст значение `2147483647`.

Преобразование при инициализации с помощью {} (C++11)

В C++11 инициализация, при которой используются фигурные скобки, называется *списковой инициализацией*. Причина в том, что эта форма позволяет предоставлять списки значений для более сложных типов данных. Это более ограничивающий тип преобразования, чем формы, используемые в листинге 3.13. В частности, списковая инициализация не допускает *сужение*, при котором тип переменной может быть не в состоянии представить присвоенной значение. Например, преобразования типов с плавающей точкой в целочисленные типы не разрешены. Преобразование из целочисленных типов в другие целочисленные типы или типы с плавающей точкой может быть разрешено, если компилятор имеет возможность сообщить, способна ли целевая переменная корректно хранить значение. Например, инициализировать переменную `long` значением `int` вполне нормально, поскольку тип `long` всегда не может превышать `int`. Преобразования в другом направлении могут быть разрешены, если значение – это константа, которая поддерживается целевым типом:

```
const int code = 66;
int x = 66;
char c1 {31325};      // сужение, не разрешено
char c2 = {66};       // разрешено, поскольку char может хранить значение 66
char c3 {code};       // то же самое
char c4 = {x};        // не разрешено, x не является константой
x = 31325;
char c5 = x;          // разрешено этой формой инициализации
```

При инициализации c4 известно, что x имеет значение 66, но для компилятора x — это переменная, которая предположительно может иметь какое-то другое, намного большее значение. В обязанности компилятора не входит отслеживание того, что может произойти с переменной x между моментом ее инициализации и моментом, когда она задействована в попытке инициализации c4.

Преобразования в выражениях

Давайте разберемся с тем, что происходит при комбинировании двух различных арифметических типов в одном выражении. В подобных случаях C++ выполняет два вида автоматических преобразований. Во-первых, некоторые типы автоматически преобразуются везде, где они встречаются. Во-вторых, некоторые типы преобразуются, когда они скомбинированы с другими типами в выражении.

Сначала рассмотрим автоматические преобразования. Когда C++ оценивает выражение, значения bool, char, unsigned char, signed char и short преобразуются в int. В частности, значение true преобразуется в 1, а false — в 0. Такие преобразования называются *целочисленными расширениями*. В качестве примера рассмотрим следующие операторы:

```
short chickens = 20;              // строка 1
short ducks = 35;                 // строка 2
short fowl = chickens + ducks;    // строка 3
```

Чтобы выполнить оператор в строке 3, программа C++ получает значения переменных chickens и ducks и преобразует их в тип int. Затем программа преобразует полученный результат обратно в short, поскольку конечный результат присваивается переменной типа short. Это может показаться хождением по кругу, однако оно имеет смысл. Для компьютера int обычно является наиболее естественным типом, поэтому все вычисления с ним могут оказаться самыми быстрыми.

Существуют и другие целочисленные расширения: тип unsigned short преобразуется в int, если тип short короче, чем int. Если оба типа имеют одинаковый размер, то unsigned short преобразуется в unsigned int. Это правило гарантирует, что никакой потери данных при расширении unsigned short не будет. Подобным образом расширяется и тип wchar_t до первого из следующих типов, который достаточно широк, чтобы уместить его диапазон: int, unsigned int, long или unsigned long.

Возможны также преобразования при арифметическом комбинировании различных типов, например, при сложении значений int и float. Если в арифметической операции участвуют два разных типа, то меньший тип преобразуется в больший. Например, в программе из листинга 3.11 значение 9.0 делится на 5. Поскольку 9.0 имеет тип double, программа, прежде чем произвести деление, преобразует значение 5 в тип double. Вообще говоря, при определении, какие преобразования необходимы в арифметическом выражении, компилятор руководствуется контрольным списком.

В C++11 этот список претерпел некоторые изменения и представлен ниже.

1. Если один из операндов имеет тип `long double`, то другой операнд преобразуется в `long double`.
2. Иначе, если один из операндов имеет тип `double`, то другой операнд преобразуется в `double`.
3. Иначе, если один из операндов имеет тип `float`, то другой операнд преобразуется в `float`.
4. Иначе, операнды имеют целочисленный тип, поэтому выполняется целочисленное расширение.
5. В этом случае, если оба операнда имеют знак или оба операнда беззнаковые, и один из них имеет меньший ранг, чем другой, он преобразуется в больший ранг.
6. Иначе, один операнд имеет знак, а другой беззнаковый. Если беззнаковый операнд имеет больший ранг, чем операнд со знаком, последний преобразует в тип беззнакового операнда.
7. Иначе, если тип со знаком может представить все значения беззнакового типа, беззнаковый операнд преобразуется к типу операнда со знаком.
8. Иначе, оба операнда преобразуются в беззнаковую версию типа со знаком.

Стандарт ANSI C следует тем же правилам, что и ISO 2003 C++, которые слегка отличаются от приведенных выше, а классическая версия K&R C имеет также немного отличающиеся правила. Например, в классическом C тип `float` всегда расширяется к `double`, даже если оба операнда относятся к типу `float`.

В этом списке была введена концепция назначения рангов целочисленным типам. Вкратце, как вы могли ожидать, базовые ранги для целочисленных типов со знаком, от большего к меньшему, выглядят следующим образом: `long long`, `long`, `int`, `short` и `signed char`. Беззнаковые типы имеют те же самые ранги, что и соответствующие им типы со знаком. Три типа — `char`, `signed char` и `unsigned char` — имеют один и тот же ранг. Тип `bool` имеет наименьший ранг. Типы `wchar_t`, `char16_t` и `char32_t` имеют те же ранги, что и типы, лежащие в их основе.

Преобразования при передаче аргументов

Обычно в C++ преобразованиями типов при передаче аргументов управляют прототипы функций, как будет показано в главе 7. Однако возможно, хотя и неблагоразумно, отказаться от управления передачей аргументов со стороны прототипа. В этом случае C++ применяет целочисленное расширение для типов `char` и `short` (`signed` и `unsigned`). Кроме того, для сохранения совместимости с большим объемом кода на классическом C, C++ расширяет аргументы `float` до `double` при передаче их функции, в которой не используется управление со стороны прототипа.

Приведение типов

C++ позволяет явно обеспечить преобразование типов через механизм приведения. (C++ признает необходимость наличия правил, связанных с типами, но также разрешает временами переопределять эти правила.) Приведение типа может быть осуществлено двумя способами. Например, чтобы преобразовать значение `int`, хранящееся в переменной по имени `thorn`, в тип `long`, можно использовать одно из следующих выражений:

```
(long) thorn          // возвращает результат преобразования thorn в тип long
long (thorn)          // возвращает результат преобразования thorn в тип long
```

Приведение типа не изменяет значение самой переменной thorn; вместо этого создается новое значение указанного типа, которое затем можно использовать в выражении, например:

```
cout << int('Q');     // отображает целочисленный код для 'Q'
```

В общих чертах можно делать следующее:

```
(имяТипа) значение    // преобразует значение в тип имяТипа
имяТипа (значение)    // преобразует значение в тип имяТипа
```

Первая форма представляет стиль C, а вторая — стиль C++. Идея новой формы заключается в том, чтобы оформить приведение типа точно так же, как и вызов функции. В результате приведения для встроенных типов будут выглядеть так же, как и преобразования типов, разрабатываемые для определяемых пользователями классов.

C++ также предлагает четыре операции приведения типов с более ограниченными возможностями применения. Они рассматриваются в главе 15. Одна из этих четырех операций, static_cast<>, может использоваться для преобразования значений из одного числового типа в другой. Например, ее можно применять для преобразования переменной thorn в значение типа long:

```
static_cast<long> (thorn)          // возвращает результат преобразования
                                   // thorn в тип long
```

В общих чертах можно делать следующее:

```
static_cast<имяТипа> (значение)    // преобразует значение в тип имяТипа
```

Как будет показано в главе 15, Страуструп был убежден, что традиционное приведение типа в стиле C опасно неограничен в своих возможностях. Операция static_cast<> является более ограниченной, чем традиционное приведение типа.

В листинге 3.14 кратко иллюстрируется использование базового приведения типа (две формы) и static_cast<>. Представьте, что первая часть этого листинга является частью мощной программы моделирования экологической ситуации, вычисления которой производятся в формате с плавающей точкой, а результат преобразуется в целочисленные значения, представляющие количество птиц и животных. Полученный результат зависит от того, в какой момент осуществляется преобразование. При вычислении auks сначала суммируются значения с плавающей точкой, и перед присваиванием сумма преобразуется в int. Однако в вычислениях bats и coots сначала используются приведения типов для преобразования значений с плавающей точкой в int, а затем полученные значения суммируются. В финальной части программы показано, как использовать приведение типа для отображения кода ASCII, соответствующего значению типа char.

Листинг 3.14. typecast.cpp

```
// typecast.cpp -- принудительное изменение типов
#include <iostream>
int main()
{
    using namespace std;
    int auks, bats, coots;

    // следующий оператор суммирует значения типа double,
    // а полученный результат преобразуется в тип int
    auks = 19.99 + 11.99;
```

```
    // эти операторы суммируют целочисленные значения
    bats = (int) 19.99 + (int) 11.99;          // старый синтаксис C
    coots = int (19.99) + int (11.99);         // новый синтаксис C++
    cout << "auks = " << auks << ", bats = " << bats;
    cout << ", coots = " << coots << endl;

    char ch = 'Z';
    cout << "The code for " << ch << " is ";   // вывод в формате char
    cout << int(ch) << endl;                   // вывод в формате int
    cout << "Yes, the code is ";
    cout << static_cast<int>(ch) << endl;      // использование static_cast
    return 0;
}
```

Вот как выглядят результаты выполнения этой программы:

```
auks = 31, bats = 30, coots = 30
The code for Z is 90
Yes, the code is 90
```

Сложение чисел `19.99` и `11.99` дает в результате `31.98`. Когда это значение присваивается переменной `auks` типа `int`, оно усекается до `31`. Однако если использовать приведения типов до суммирования, то значения будут усечены до `19` и `11`, поэтому в результате переменные `bats` и `coots` получат значение `30`. Затем в двух операторах `cout` приведения типов применяются для преобразования значений `char` в `int` перед их отображением. Эти преобразования приводят к тому, что `cout` выведет значение в виде целого числа, а не символа.

В этой программе проиллюстрированы две причины использования приведения типов. Скажем, у вас могут быть значения, которые хранятся в формате `double`, но используются для вычисления значения `int`. Например, требуется привязка к позиции на сетке или моделирование целочисленных значений, таких как размер популяции, с помощью значений с плавающей точкой. Может понадобиться, чтобы в вычислениях все значения трактовались как `int`. Все это позволяет сделать приведение типов. Обратите внимание, что вы получите разные результаты (во всяком случае, для этих значений) в ситуациях, когда сначала применяется преобразование в `int` и затем суммирование, и когда сначала выполняется суммирование, а затем преобразование в `int`.

Во второй части программы продемонстрирована наиболее распространенная причина использования приведения типа: возможность заставить данные в одной форме удовлетворять различным ожиданиям. Например, в листинге 3.14 переменная `ch` типа `char` хранит код буквы `Z`. Использование `cout` для вывода `ch` приводит к отображению буквы `Z`, поскольку `cout` концентрирует внимание на факте принадлежности переменной `ch` к типу `char`. Однако за счет приведения `ch` к типу `int` объект `cout` переключается на режим `int` и выводит ASCII-код, хранящийся в `ch`.

Объявления `auto` в C++11

В C++11 появилось средство, которое позволяет компилятору выводить тип из типа значения инициализации. Для этой цели было переопределено назначение `auto` — ключевого слова, восходящего к временам C, но почти не используемого. (Предыдущее назначение `auto` описано в главе 9.) Просто укажите `auto` вместо имени типа в инициализирующем объявлении, и компилятор назначает переменной тот же самый тип, что у инициализатора:

```
auto n = 100;          // n получает тип int
auto x = 1.5;          // x получает тип double
auto y = 1.3e12L;      // y получает тип long double
```

Однако это автоматическое выведение типа на самом деле не предназначено для таких простых случаев. В действительности можно даже запутаться. Например, предположим, что x, y и z должны относиться к типу double. Взгляните на следующий код:

```
auto x = 0.0;    // нормально, x является double, поскольку 0.0 - это double
double y = 0;    // нормально, 0 автоматически преобразуется в 0.0
auto z = 0;      // проблема, z является int, поскольку 0 - это int
```

Использование 0 вместо 0.0 не вызывает проблем при явном указании типов, но приводит к ним в случае автоматического преобразования типов.

Автоматическое выведение типа становится более полезным, когда приходится иметь дело со сложными типами, такими как применяемые в стандартной библиотеке шаблонов (STL). Например, код на C++98 может содержать следующие операторы:

```
std::vector<double> scores;
std::vector<double>::iterator pv = scores.begin();
```

C++11 позволяет вместо них записать так:

```
std::vector<double> scores;
auto pv = scores.begin();
```

Об этом новом назначении ключевого слова auto еще пойдет речь позже, в более подходящем контексте.

Резюме

Базовые типы в C++ делятся на две группы. Одна группа представляет значения, которые хранятся в виде целых чисел, а другая группа — значения, хранящиеся в формате с плавающей точкой. Целочисленные типы отличаются друг от друга по количеству памяти, которое отводится для хранения значений, а также по наличию знака. Целочисленными типами являются следующие (от меньших к большим): bool, char, signed char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, а также появившиеся в C++11 типы long long и unsigned long long. Существует также тип wchar_t, место которого в приведенной последовательности зависит от реализации. В C++11 добавлены типы char16_t и char32_t, которые имеют ширину, достаточную для представления 16- и 32-битных кодов символов, соответственно. Язык C++ гарантирует, что тип char имеет достаточно большой размер, чтобы хранить любой член расширенного набора символов в системе, тип short имеет как минимум 16 битов, int как минимум такой же, как short, а long имеет минимум 32 бита и является как минимум таким же, как int. Точные размеры зависят от реализации.

Символы представлены своими числовыми кодами. Система ввода-вывода определяет, как интерпретируется код — как символ или как число.

Типы с плавающей точкой могут представлять дробные значения и значения намного больше тех, которые могут быть представлены целочисленными типами. Типов с плавающей точкой всего три: float, double и long double. C++ гарантирует, что тип float не меньше, чем тип double, и что тип double не больше типа long double. Обычно тип float использует 32 бита памяти, double — 64 бита, а long double — от 80 до 128 битов.

Предоставляя широкое разнообразие типов с разными размерами и версиями со знаком и без знака, C++ позволяет найти тип, который в точности удовлетворяет конкретным требованиям к данным.

Над числовыми типами в C++ можно выполнять следующие арифметические операции: сложение, вычитание, умножение, деление и нахождение остатка от деления. Когда две операции добиваются применения к одному значению, выбор той из них, которая будет выполнена первой, осуществляется в соответствии с правилами приоритетов и ассоциативности.

C++ преобразует значение одного типа в другой, когда вы присваиваете значение переменной, комбинируете разные типы в арифметических операциях и используете приведение для принудительного преобразования типов. Многие преобразования типов являются "безопасными", т.е. они могут быть выполнены без потери или изменения данных. Например, вы можете без проблем преобразовать значение `int` в `long`. Другие преобразования, например, преобразование типов с плавающей точкой в целочисленные типы, требуют большей осторожности.

На первый взгляд может показаться, что в C++ определено слишком много базовых типов, особенно если принимать во внимание различные правила преобразования. Однако подобное разнообразие типов позволяет выбрать именно такой тип, который наиболее полно соответствует существующим требованиям.

Вопросы для самоконтроля

1. Почему в языке C++ имеется более одного целочисленного типа?
2. Объявите переменные согласно перечисленным ниже описаниям.
   а. Целочисленная переменная `short`, имеющая значение 80.
   б. Целочисленная переменная `unsigned int`, имеющая значение 42.110.
   в. Целочисленная переменная, имеющая значение 3 000 000 000.
3. Какие меры предпринимаются в C++, чтобы не допустить превышения пределов целочисленного типа?
4. В чем состоит различие между `33L` и `33`?
5. Взгляните на следующие два оператора C++:

   ```
   char grade = 65;
   char grade = 'A';
   ```

   Являются ли они эквивалентными?
6. Как в C++ определить, какой символ представляется кодом 88? Сделайте это, по крайней мере, двумя способами.
7. Присваивание значения типа `long` переменной типа `float` может привести к ошибке округления. А что произойдет, если присвоить значение `long` переменной `double`? И что будет, если присвоить значение `long long` переменной `double`?
8. Вычислите следующие выражения:

   а. `8 * 9 + 2`
   б. `6 * 3 / 4`
   в. `3 / 4 * 6`
   г. `6.0 * 3 / 4`
   д. `15 % 4`

9. Предположим, что x1 и x2 являются переменными типа double, которые вы хотите просуммировать как целые числа, а полученный результат присвоить целочисленной переменной. Напишите для этого необходимый оператор C++. Что если вы хотите просуммировать их как значения double, а затем преобразовать результат в int?
10. Каким будет тип переменной в каждом из следующих объявлений?

    а. `auto cars = 15;`

    б. `auto iou = 150.37f;`

    в. `auto level = 'B';`

    г. `auto crat = U'/U00002155';`

    д. `auto fract = 8.25f/2.5;`

Упражнения по программированию

1. Напишите короткую программу, которая запрашивает рост в дюймах и преобразует их в футы и дюймы. Программа должна использовать символ подчеркивания для обозначения позиции, где будет производиться ввод. Для представления коэффициента преобразования используйте символьную константу const.
2. Напишите короткую программу, которая запрашивает рост в футах и дюймах и вес в фунтах. (Для хранения этой информации используйте три переменных.) Программа должна выдать индекс массы тела (body mass index – BMI). Чтобы рассчитать BMI, сначала преобразуйте рост в футах и дюймах в рост в дюймах (1 фут = 12 дюймов). Затем преобразуйте рост в дюймах в рост в метрах, умножив на 0.0254. Далее преобразуйте вес в фунтах в массу в килограммах, разделив на 2.2. После этого рассчитайте BMI, разделив массу в килограммах на квадрат роста в метрах. Для представления различных коэффициентов преобразования используйте символические константы.
3. Напишите программу, которая запрашивает широту в градусах, минутах и секундах, после чего отображает широту в десятичном формате. В одной минуте 60 угловых секунд, а в одном градусе 60 угловых минут; представьте эти значения с помощью символических констант. Для каждого вводимого значения должна использоваться отдельная переменная. Результат выполнения программы должен выглядеть следующим образом:

```
Enter a latitude in degrees, minutes, and seconds:
First, enter the degrees: 37
Next, enter the minutes of arc: 51
Finally, enter the seconds of arc: 19
37 degrees, 51 minutes, 19 seconds = 37.8553 degrees
```

4. Напишите программу, которая запрашивает количество секунд в виде целого значения (используйте тип long или long long, если последний доступен) и затем отображает эквивалентное значение в сутках, часах, минутах и секундах. Для представления количества часов в сутках, количества минут в часе и количества секунд в минуте используйте символические константы. Результат выполнения программы должен выглядеть следующим образом:

```
Enter the number of seconds: 31600000
31600000 seconds = 365 days, 17 hours, 46 minutes, 40 seconds
```

5. Напишите программу, которая запрашивает текущую численность населения Земли и текущую численность населения США (или любой другой страны). Сохраните эту информацию в переменных типа `long long`. В качестве результата программа должна отображать процентное соотношение численности населения США (или выбранной страны) и всего мира. Результат выполнения программы должен выглядеть следующим образом:

```
Enter the world's population: 6898758899
Enter the population of the US: 310783781
The population of the US is 4.50492% of the world population.
```

   Можете поискать в Интернете более точные значения.

6. Напишите программу, которая запрашивает количество миль, пройденных автомобилем, и количество галлонов израсходованного бензина, а затем сообщает значение количества миль на галлон. Или, если хотите, программа может запрашивать расстояние в километрах, а объем бензина в литрах, и выдавать результат в виде количества литров на 100 километров.

7. Напишите программу, которая запрашивает расход бензина в европейском стиле (количество литров на 100 км) и преобразует его в стиль, принятый в США — число миль на галлон. Обратите внимание, что кроме использования других единиц измерений, принятый в США подход (расстояние/топливо) противоположен европейскому (топливо/расстояние). Учтите, что 100 километров соответствуют 62.14 милям, а 1 галлон составляет 3.875 литра. Таким образом, 19 миль на галлон примерно равно 12.4 литров на 100 км, а 27 миль на галлон — примерно 8.7 литров на 100 км.

4

Составные типы

В ЭТОЙ ГЛАВЕ...

- Создание и использование массивов
- Создание и использование строк в стиле C
- Создание и использование строк класса `string`
- Использование методов `get()` и `getline()` для чтения строк
- Смешивание строкового и числового ввода
- Создание и использование структур
- Создание и использование объединений
- Создание и использование перечислений
- Создание и использование указателей
- Управление динамической памятью с помощью `new` и `delete`
- Создание динамических массивов
- Создание динамических структур
- Автоматическое, статическое и динамическое хранилище
- Классы `vector` и `array` (введение)

Предположим, что вы разработали компьютерную игру под названием "Враждебный пользователь", в которой игроки состязаются с замысловатым и недружественным компьютерным интерфейсом. Теперь вам необходимо написать программу, которая отслеживает ежемесячные объемы продаж этой игры в течение пятилетнего периода. Или, скажем, вам нужно провести инвентаризацию торговых карт героев-хакеров. Очень скоро вы придете к выводу, что для накопления и обработки информации вам требуется нечто большее, чем простые базовые типы C++. И C++ предлагает это нечто большее, а именно — составные типы. Это типы, состоящие из базовых целочисленных типов и типов с плавающей точкой. Наиболее развитым составным типом является класс — оплот объектно-ориентированного программирования (ООП), к которому мы стремимся двигаться. Но C++ также поддерживает несколько более скромных составных типов, которые взяты из языка C. Массив, например, может хранить множество значений одного и того же типа. Отдельный вид массива может хранить строки, которые являются последовательностями символов. Структуры могут хранить по нескольку значений разных типов. Кроме того, есть еще указатели, которые представляют собой переменные, сообщающие компьютеру местонахождение данных в памяти. Все эти составные формы данных (кроме классов) мы рассмотрим в настоящей главе. Вы кратко ознакомитесь с операциями `new` и `delete`, а также получите первое представление о классе C++ по имени `string`, который предлагает альтернативный способ работы со строками.

Введение в массивы

Массив — это структура данных, которая содержит множество значений, относящихся к одному и тому же типу. Например, массив может содержать 60 значений типа `int`, которые представляют информацию об объемах продаж за 5 лет, 12 значений типа `short`, представляющих количество дней в каждом месяце, или 365 значений типа `float`, которые указывают ежедневные расходы на питание в течение года. Каждое значение сохраняется в отдельном элементе массива, и компьютер хранит все элементы массива в памяти последовательно — друг за другом.

Для создания массива используется оператор объявления. Объявление массива должно описывать три аспекта:

- тип значений каждого элемента;
- имя массива;
- количество элементов в массиве.

В C++ это достигается модификацией объявления простой переменной, к которому добавляются квадратные скобки, содержащие внутри количество элементов. Например, следующее объявление создает массив по имени `months`, имеющий 12 элементов, каждый из которых может хранить одно значение типа `short`:

```
short months[12];    // создает массив из 12 элементов типа short
```

В сущности, каждый элемент — это переменная, которую можно трактовать как простую переменную.

Так выглядит общая форма объявления массива:

```
имяТипа имяМассива[размерМассива];
```

Выражение *размерМассива*, представляющее количество элементов, должно быть целочисленной константой, такой как 10, значением `const` либо константным выражением вроде `8 * sizeof(int)`, в котором все значения известны на момент компи-

ляции. В частности, *размерМассива* не может быть переменной, значение которой устанавливается во время выполнения программы. Однако позднее в этой главе вы узнаете, как с использованием операции `new` обойти это ограничение.

Массив как составной тип

Массив называют *составным типом*, потому что он строится из какого-то другого типа. (В языке C используется термин *производный тип*, но поскольку понятие *производный* в C++ применяется для описания отношений между классами, пришлось ввести новый термин.) Вы не можете просто объявить, что нечто является массивом; это всегда должен быть массив элементов конкретного типа. Обобщенного типа массива не существует. Вместо этого имеется множество специфических типов массивов, таких как массив `char` или массив `long`. Например, рассмотрим следующее объявление:

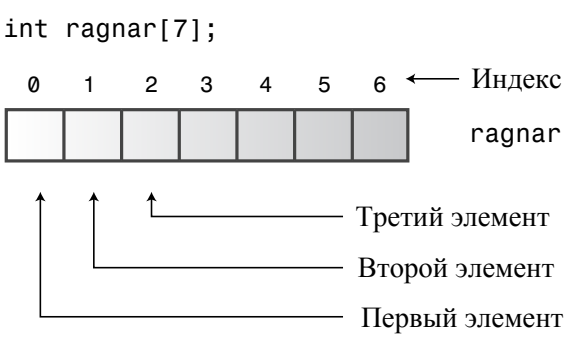
```
float loans[20];
```

Типом переменной `loans` будет не просто "массив", а "массив `float`". Это подчеркивает, что массив `loans` построен из типа `float`.

Большая часть пользы от массивов определяется тем фактом, что к его элементам можно обращаться индивидуально. Способ, который позволяет это делать, заключается в использовании *индекса* для нумерации элементов. Нумерация массивов в C++ начинается с нуля. (Это является обязательным — вы должны начинать с нуля. Это особенно важно запомнить программистам, ранее работавшим на языках Pascal и BASIC.) Для указания элемента массива в C++ используется обозначение с квадратными скобками и индексом между ними. Например, `months[0]` — это первый элемент массива `months`, а `months[11]` — его последний элемент. Обратите внимание, что индекс последнего элемента на единицу меньше, чем размер массива (рис. 4.1). Таким образом, объявление массива позволяет создавать множество переменных в одном объявлении, и вы затем можете использовать индекс для идентификации и доступа к индивидуальным элементам.

Важность указания правильных значений

Компилятор не проверяет правильность указываемого индекса. Например, компилятор не станет жаловаться, если вы присвоите значение несуществующему элементу `months[101]`. Однако такое присваивание может вызвать проблемы во время выполнения программы — возможно, повреждение данных или кода, а может быть и аварийное завершение программы. То есть обеспечение правильности значений индекса возлагается на программиста.



Массив, хранящий семь значений, каждое из которых является переменной типа `int`

Рис. 4.1. *Создание массива*

Небольшая программа анализа, представленная в листинге 4.1, демонстрирует несколько свойств массивов, включая их объявление, присваивание значение его элементам, а также инициализацию.

Листинг 4.1. arrayone.cpp

```
// arrayone.cpp -- небольшие массивы целых чисел
#include <iostream>
int main()
{
    using namespace std;
    int yams[3];                    // создание массива из трех элементов
    yams[0] = 7;                    // присваивание значения первому элементу
    yams[1] = 8;
    yams[2] = 6;

    int yamcosts[3] = {20, 30, 5};  // создание и инициализация массива
    // Примечание. Если ваш компилятор C++ не может инициализировать
    // этот массив, используйте static int yamcosts[3] вместо int yamcosts[3]

    cout << "Total yams = ";
    cout << yams[0] + yams[1] + yams[2] << endl;
    cout << "The package with " << yams[1] << " yams costs ";
    cout << yamcosts[1] << " cents per yam.\n";
    int total = yams[0] * yamcosts[0] + yams[1] * yamcosts[1];
    total = total + yams[2] * yamcosts[2];
    cout << "The total yam expense is " << total << " cents.\n";

    cout << "\nSize of yams array = " << sizeof yams;
    cout << " bytes.\n";
    cout << "Size of one element = " << sizeof yams[0];
    cout << " bytes.\n";
    return 0;
}
```

Ниже показан вывод программы из листинга 4.1:

```
Total yams = 21
The package with 8 yams costs 30 cents per yam.
The total yam expense is 410 cents.

Size of yams array = 12 bytes.
Size of one element = 4 bytes.
```

Замечания по программе

Сначала программа в листинге 4.1 создает массив из трех элементов по имени `yams`. Поскольку `yams` имеет три элемента, они нумеруются от 0 до 2, и `arrayone.cpp` использует значения индекса от 0 до 2 для присваивания значений трем отдельным элементам. Каждый индивидуальный элемент `yams` — это переменная типа `int`, со всеми правилами и привилегиями типа `int`, поэтому `arrayone.cpp` может (и делает это) присваивать значения элементам, складывать элементы, перемножать их и отображать.

В этой программе применяется длинный способ присваивания значений элементам `yams`. C++ также позволяет инициализировать элементы массива непосредственно в операторе объявления. В листинге 4.1 демонстрируется этот сокращенный способ при установке значений элементов массива `yamcosts`:

```
int yamcosts[3] = {20, 30, 5};
```

Он просто предоставляет разделенный запятыми список значений (*список инициализации*), заключенный в фигурные скобки. Пробелы в списке не обязательны. Если вы не инициализируете массив, объявленный внутри функции, его элементы остаются неопределенными. Это значит, что элементы получают случайные значения, которые зависят от предыдущего содержимого области памяти, выделенной для такого массива.

Далее программа использует значения массива в нескольких вычислениях. Эта часть программы выглядит несколько беспорядочно со всеми этими индексами и скобками. Цикл `for`, который будет описан в главе 5, предоставляет мощный способ работы с массивами и исключает необходимость явного указания индексов. Но пока мы ограничимся небольшими массивами.

Как вы, возможно, помните, операция `sizeof` возвращает размер в байтах типа или объекта данных. Обратите внимание, что применение `sizeof` к имени массива дает количество байт, занимаемых всем массивом. Однако использование `sizeof` в отношении элемента массива дает размер в байтах одного элемента. Это иллюстрирует тот факт, что `yams` — массив, но `yams[1]` — просто `int`.

Правила инициализации массивов

В C++ существует несколько правил, касающихся инициализации массивов. Они ограничивают, когда вы можете ее осуществлять, и определяют, что случится, если количество элементов массива не соответствует количеству элементов инициализатора. Давайте рассмотрим эти правила.

Вы можете использовать инициализацию *только* при объявлении массива. Ее нельзя выполнить позже, и нельзя присваивать один массив другому:

```
int cards[4] = {3, 6, 8, 10};      // все в порядке
int hand[4];                       // все в порядке
hand[4] = {5, 6, 7, 9};            // не допускается
hand = cards;                      // не допускается
```

Однако можно использовать индексы и присваивать значения элементам массива индивидуально.

При инициализации массива можно указывать меньше значений, чем в массиве объявлено элементов. Например, следующий оператор инициализирует только первые два элемента массива `hotelTips`:

```
float hotelTips[5] = {5.0, 2.5};
```

Если вы инициализируете массив частично, то компилятор присваивает остальным элементом нулевые значения. Это значит, что инициализировать весь массив нулями очень легко — для этого просто нужно явно инициализировать нулем его первый элемент, а инициализацию остальных элементов поручить компилятору:

```
long totals[500] = {0};
```

Следует отметить, что в случае инициализации массива с применением `{1}` вместо `{0}` только первый элемент будет установлен в 1; остальные по-прежнему получат значение 0.

Если при инициализации массива оставить квадратные скобки пустыми, то компилятор C++ самостоятельно пересчитает элементы. Предположим, например, что есть следующее объявление:

```
short things[] = {1, 5, 3, 8};
```

Компилятор сделает `things` массивом из пяти элементов.

Позволять ли компилятору самостоятельно подсчитывать элементы?

Часто компилятор при подсчете элементов получает не то количество, которое, как вы ожидаете, должно быть. Причиной может быть, например, непреднамеренный пропуск одного или нескольких значений в списке инициализации. Однако, как вы вскоре убедитесь, такой подход может быть вполне безопасным для инициализации символьного массива строкой. И если главная цель состоит в том, чтобы программа, а не вы, знала размер массива, то можно записать примерно следующий код:

```
short things[] = {1, 5, 3, 8};
int num_elements = sizeof things / sizeof (short);
```

Удобно это или нет — зависит от сложившихся обстоятельств.

Инициализация массивов в C++11

Как упоминалось в главе 3, в C++11 форма инициализации с фигурными скобками (списковая инициализация) стала универсальной для всех типов. Массивы уже используют списковую инициализацию, но в версии C++11 появились дополнительные возможности.

Во-первых, при инициализации массива можно отбросить знак =:

```
double earnings[4] {1.2e4, 1.6e4, 1.1e4, 1.7e4}; // допускается в C++11
```

Во-вторых, можно использовать пустые фигурные скобки для установки всех элементов в 0:

```
unsigned int counts[10] = {};           // все элементы устанавливаются в 0
float balances[100] {};                 // все элементы устанавливаются в 0
```

В-третьих, как обсуждалось в главе 3, списковая инициализация защищает от сужения:

```
long plifs[] = {25, 92, 3.0};           // не разрешено
char slifs[4] {'h', 'i', 1122011, '\0'}; // не разрешено
char tlifs[4] {'h', 'i', 112, '\0'};    // разрешено
```

Первая инициализация не допускается, т.к. преобразование из типа с плавающей точкой в целочисленный тип является сужением, даже если значение с плавающей точкой содержит после десятичной точки только нули. Вторая инициализация не допускается, поскольку `1122011` выходит за пределы диапазона значений типа `char`, предполагая, что `char` занимает 8 бит. Третья инициализация выполняется успешно, т.к. несмотря на то, что `112` является значением `int`, оно находится в рамках диапазона типа `char`.

Стандартная библиотека шаблонов C++ (STL) предлагает альтернативу массивам — шаблонный класс `vector`, а в C++11 еще добавлен шаблонный класс `array`. Эти альтернативы являются более сложными и гибкими, нежели встроенный составной тип массива. Они кратко будут рассматриваться далее в этой главе и более подробно — в главе 16.

Строки

Строка — это серия символов, сохраненная в расположенных последовательно байтах памяти. В C++ доступны два способа работы со строками. Первый, унаследованный от C и часто называемый *строками в стиле C*, рассматривается в настоящей главе сначала. Позже будет описан альтернативный способ, основанный на библиотечном классе `string`.

Идея серии символов, сохраняемых в последовательных байтах, предполагает хранение строки в массиве char, где каждый элемент содержится в отдельном элементе массива. Строки предоставляют удобный способ хранения текстовой информации, такой как сообщения для пользователя или его ответы. Строки в стиле C обладают специальной характеристикой: последним в каждой такой строке является *нулевой символ*. Этот символ, записываемый как \0, представляет собой символ с ASCII-кодом 0, который служит меткой конца строки. Например, рассмотрим два следующих объявления:

```
char dog[8] = { 'b', 'e', 'a', 'u', 'x', ' ', 'I', 'I'};    // это не строка
char cat[8] = {'f', 'a', 't', 'e', 's', 's', 'a', '\0'};    // а это — строка
```

Обе эти переменные представляют собой массивы char, но только вторая из них является строкой. Нулевой символ играет фундаментальную роль в строках стиля C. Например, в C++ имеется множество функций для обработки строк, включая те, что используются cout. Все они обрабатывают строки символ за символом до тех пор, пока не встретится нулевой символ. Если вы просите объект cout отобразить такую строку, как cat из предыдущего примера, он выводит первых семь символов, обнаруживает нулевой символ и на этом останавливается. Однако если вы вдруг решите вывести в cout массив dog из предыдущего примера, который не является строкой, то cout напечатает восемь символов из этого массива и будет продолжать двигаться по памяти, байт за байтом, интерпретируя каждый из них как символ, подлежащий выводу, пока не встретит нулевой символ. Поскольку нулевые символы, которые, по сути, представляют собой байты, содержащие нули, встречаются в памяти довольно часто, ошибка обычно обнаруживается быстро, но в любом случае вы не должны трактовать нестроковые символьные массивы как строки.

Пример инициализации массива cat выглядит довольно громоздким и утомительным — множество одиночных кавычек плюс необходимость помнить о нулевом символе. Не волнуйтесь. Существует более простой способ инициализации массива с помощью строки. Для этого просто используйте строку в двойных кавычках, которая называется *строковой константой* или *строковым литералом*, как показано ниже:

```
char bird[11] = "Mr. Cheeps";      // наличие символа \0 подразумевается
char fish[] = "Bubbles";           // позволяет компилятору подсчитать
                                   // количество элементов
```

Строки в двойных кавычках всегда неявно включают ограничивающий нулевой символ, поэтому указывать его явно не требуется (рис. 4.2.) К тому же разнообразные средства ввода C++, предназначенные для чтения строки с клавиатурного ввода в массив char, автоматически добавляют завершающий нулевой символ. (Если при компиляции программы из листинга 4.1 вы обнаружите необходимость в использовании ключевого слова static для инициализации массива, это также понадобится сделать с показанными выше массивами char.)

```
char boss[8] = "Bozo";
```

***Рис. 4.2.** Инициализация массива строкой*

Разумеется, вы должны обеспечить достаточный размер массива, чтобы в него поместились все символы строки, включая нулевой. Инициализация символьного массива строковой константой — это один из тех случаев, когда безопаснее поручить компилятору подсчет количества элементов в массиве. Если сделать массив больше строки, никаких проблем не возникнет — только непроизводительный расход пространства. Причина в том, что функции, которые работают со строками, руководствуются позицией нулевого символа, а не размером массива. В C++ не накладывается никаких ограничений на длину строки.

На заметку!

При определении минимального размера массива, необходимого для хранения строки, не забудьте учесть при подсчете завершающий нулевой символ.

Обратите внимание, что строковая константа (в двойных кавычках) не взаимозаменяема с символьной константой (в одинарных кавычках). Символьная константа, такая как `'S'`, представляет собой сокращенное обозначение для кода символа. В системе ASCII константа `'S'` — это просто другой способ записи кода 83. Поэтому следующий оператор присваивает значение 83 переменной `shirt_size`:

```
char shirt_size = 'S';    // нормально
```

С другой стороны, `"S"` не является символьной константой; это строка, состоящая из двух символов — `S` и `\0`. Хуже того, `"S"` в действительности представляет адрес памяти, по которому размещается строка. Это значит, что приведенный ниже оператор означает попытку присвоить адрес памяти переменной `shirt_size`:

```
char shirt_size = "S";    // не допускается по причине несоответствия типов
```

Поскольку адрес памяти — это отдельный тип в C++, компилятор не пропустит подобную бессмыслицу. (Мы вернемся к этому моменту позже в данной главе, во время рассмотрения указателей.)

Конкатенация строковых литералов

Иногда строки могут оказаться слишком большими, чтобы удобно разместиться в одной строке кода. C++ позволяет выполнять конкатенацию строковых литералов — т.е. комбинировать две строки с двойными кавычками в одну. В действительности любые две строковые константы, разделенные только пробельным символом (пробелами, символами табуляции и символами новой строки), автоматически объединяются в одну. Таким образом, следующие три оператора вывода эквивалентны:

```
cout << "I'd give my right arm to be" " a great violinist.\n";
cout << "I'd give my right arm to be a great violinist.\n";
cout << "I'd give my right ar"
"m to be a great violinist.\n";
```

Обратите внимание, что такие объединения не добавляют никаких пробелов к объединяемым строкам. Первый символ второй строки немедленно следует за последним символом первой, не считая `\0` в первой строке. Символ `\0` из первой строки заменяется первым символом второй строки.

Использование строк в массивах

Два наиболее распространенных метода помещения строки в массив заключаются в инициализации массива строковой константой и чтением из клавиатурного или файлового ввода в массив. В листинге 4.2 эти подходы демонстрируются за счет ини-

циализации одного массива строкой в двойных кавычках и использования cin для помещения вводимой строки в другой массив. В программе также применяется стандартная библиотечная функция stlen() для получения длины строки. Стандартный заголовочный файл cstring (или string.h в более старых реализациях) предоставляет объявления для этой и многих других функций, работающих со строками.

Листинг 4.2. strings.cpp

```
// strings.cpp -- сохранение строк в массиве
#include <iostream>
#include <cstring>                        // для функции strlen()
int main()
{
   using namespace std;
   const int Size = 15;
   char name1[Size];                      // пустой массив
   char name2[Size] = "C++owboy";         // инициализация массива
   // ПРИМЕЧАНИЕ: некоторые реализации могут потребовать
   // ключевого слова static для инициализации массива name2

   cout << "Howdy! I'm " << name2;
   cout << "! What's your name?\n";
   cin >> name1;
   cout << "Well, " << name1 << ", your name has ";
   cout << strlen(name1) << " letters and is stored\n";
   cout << "in an array of " << sizeof(name1) << " bytes.\n";
   cout << "Your initial is " << name1[0] << ".\n";
   name2[3] = '\0';                        // установка нулевого символа
   cout << "Here are the first 3 characters of my name: ";
   cout << name2 << endl;
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.2:

```
Howdy! I'm C++owboy! What's your name?
Basicman
Well, Basicman, your name has 8 letters and is stored
in an array of 15 bytes.
Your initial is B.
Here are the first 3 characters of my name: C++
```

Замечания по программе

Чему учит код в листинге 4.2? Первым делом, обратите внимание, что операция sizeof возвращает размер всего массива — 15 байт, но функция strlen() возвращает размер строки, хранящейся в массиве, а не размер самого массива. К тому же strlen() подсчитывает только видимые символы, без нулевого символа-ограничителя. То есть эта функция возвращает в качестве длины Basicman значение 8, а не 9. Если cosmic представляет собой строку, то минимальный размер массива для размещения этой строки вычисляется как strlen(cosmic) + 1.

Поскольку name1 и name2 — массивы, для доступа к отдельным символам в этих массивах можно использовать индексы. Например, в программе для поиска первого символа массива name1 применяется name1[0]. Кроме того, программа присваивает элементу name2[3] нулевой символ. Это завершает строку после трех символов, хотя в массиве остаются еще символы (рис. 4.3).

```
const int ArSize = 15;
char name2[ArSize] = "C++owboy";
```

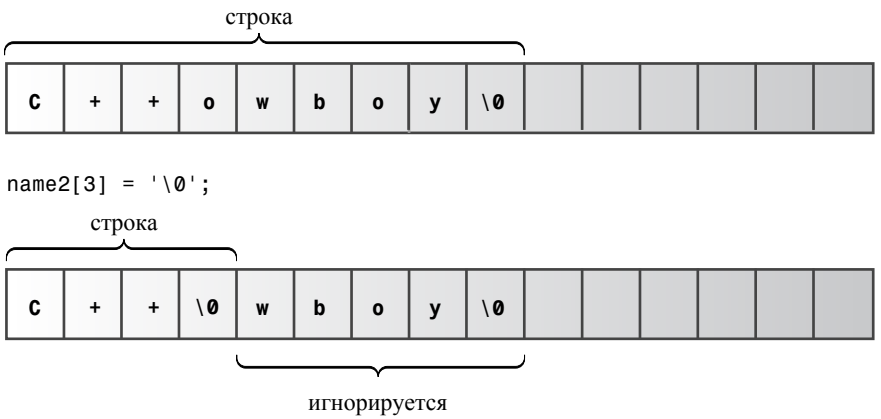


***Рис. 4.3.** Сокращение строки с помощью \0*

Обратите внимание, что в программе из листинга 4.2 для указания размера массива используется символическая константа. Часто размер массива нужно указывать в нескольких операторах программы. Применение символических констант для представления размера массива упрощает внесение изменений, связанных с длиной массива; в таких случаях изменить размер потребуется только в одном месте — там, где определена символическая константа.

Риски, связанные с вводом строк

Программа `string.cpp` имеет недостаток, скрытый за часто используемой в литературе техникой тщательного выбора примеров ввода. В листинге 4.3 демонстрируется тот факт, что строковый ввод может оказаться непростым.

Листинг 4.3. `insrt1.cpp`

```
// insrtl.cpp -- чтение более одной строки
#include <iostream>
int main()
{
   using namespace std;
   const int ArSize = 20;
   char name[ArSize];
   char dessert[ArSize];

   cout << "Enter your name:\n";                  // запрос имени
   cin >> name;
   cout << "Enter your favorite dessert:\n";      // запрос любимого десерта
   cin >> dessert;
   cout << "I have some delicious " << dessert;
   cout << " for you, " << name << ".\n";
   return 0;
}
```

Назначение программы из листинга 4.3 простое: прочитать имя пользователя и название его любимого десерта, введенные с клавиатуры, и затем отобразить эту информацию.

Ниже приведен пример запуска:

```
Enter your name:
Alistair Dreeb
Enter your favorite dessert:
I have some delicious Dreeb for you, Alistair.
```

Мы даже не получили возможности ответить на вопрос о десерте! Программа показала вопрос и затем немедленно перешла к отображению заключительной строки.

Проблема связана с тем, как `cin` определяет, когда ввод строки завершен. Вы не можете ввести нулевой символ с клавиатуры, поэтому `cin` требуется что-то другое для нахождения конца строки. Подход, принятый в `cin`, заключается в использовании пробельных символов для разделения строк — пробелов, знаков табуляции и символов новой строки. Это значит, что `cin` читает только одно слово, когда получает ввод для символьного массива. После чтения слова `cin` автоматически добавляет ограничивающий нулевой символ при помещении строки в массив.

Практический результат этого примера заключается в том, что `cin` читает слово `Alistair` как полную первую строку и помещает его в массив `name`. При этом второе слово, `Dreeb`, остается во входной очереди. Когда `cin` ищет ввод, отвечающий на вопрос о десерте, он находит там `Dreeb`. Затем `cin` захватывает слово `Dreeb` и помещает его в массив `dessert` (рис. 4.4).

Еще одна проблема, которая не была обнаружена в примере запуска, состоит в том, что вводимая строка, в свою очередь, может быть длиннее, чем целевой массив. При таком использовании `cin`, как это сделано в примере, нет никакой защиты от помещения 30-символьной строки в 20-символьный массив.

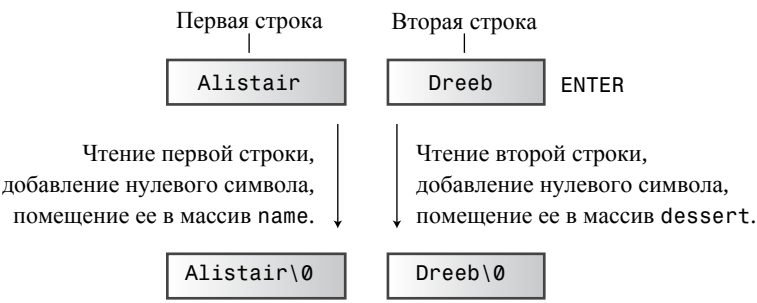


Рис. 4.4. *Строковый ввод с точки зрения* `cin`

Многие программы зависят от строкового ввода, поэтому нам стоит рассмотреть данную тему глубже. Некоторые из наиболее усовершенствованных средств `cin` будут подробно описаны в главе 17.

Построчное чтение ввода

Чтение строкового ввода по одному слову за раз — часто не является желательным поведением. Например, предположим, что программа запрашивает у пользователя ввод города, и пользователь отвечает вводом **`New York`** или **`Sao Paulo`**. Вы бы хотели, чтобы программа прочитала и сохранила полные названия, а не только `New` и `Sao`. Чтобы иметь возможность вводить целые фразы вместо отдельных слов, необходим другой подход к строковому вводу. Точнее говоря, нужен метод, ориентированный на строки, вместо метода, ориентированного на слова. К счастью, у класса `istream`, экземпляром которого является `cin`, есть функции-члены, предназначен-

ные для строчно-ориентированного ввода: `getline()` и `get()`. Оба читают полную строку ввода — т.е. вплоть до символа новой строки. Однако `getline()` затем отбрасывает символ новой строки, в то время как `get()` оставляет его во входной очереди. Давайте рассмотрим их детально, начиная с `getline()`.

Строчно-ориентированный ввод с помощью `getline()`

Функция `getline()` читает целую строку, используя символ новой строки, который передан клавишей <Enter>, для обозначения конца ввода. Этот метод инициируется вызовом функции `cin.getline()`. Функция принимает два аргумента. Первый аргумент — это имя места назначения (т.е. массива, который сохраняет введенную строку), а второй — максимальное количество символов, подлежащих чтению. Если, скажем, установлен предел 20, то функция читает не более 19 символов, оставляя место для автоматически добавляемого в конец нулевого символа. Функция-член `getline()` прекращает чтение, когда достигает указанного предела количества символов или когда читает символ новой строки — смотря, что произойдет раньше.

Например, предположим, что вы хотите воспользоваться `getline()` для чтения имени в 20-элементный массив `name`. Для этого следует указать такой вызов:

```
cin.getline(name,20);
```

Он читает полную строку в массив `name`, предполагая, что строка состоит не более чем из 19 символов. (Функция-член `getline()` также принимает необязательный третий аргумент, который обсуждается в главе 17.)

В листинге 4.4 представлен модифицированный пример из листинга 4.3 с применением `cin.getline()` вместо простого `cin`. В остальном программа осталась прежней.

Листинг 4.4. `insrt2.cpp`

```
// insrt2.cpp -- чтение более одного слова с помощью getline
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";                  // запрос имени
    cin.getline(name, ArSize);                     // читать до символа новой строки
    cout << "Enter your favorite dessert:\n";      // запрос любимого десерта
    cin.getline(dessert, ArSize);
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.4:

```
Enter your name:
Dirk Hammernose
Enter your favorite dessert:
Radish Torte
I have some delicious Radish Torte for you, Dirk Hammernose.
```

Теперь программа читает полное имя и название блюда. Функция `getline()` удобным образом принимает по одной строке за раз. Она читает ввод до новой символа строки, помечая конец строки, но не сохраняя при этом сам символ новой строки. Вместо этого она заменяет его нулевым символом при сохранении строки (рис. 4.5).

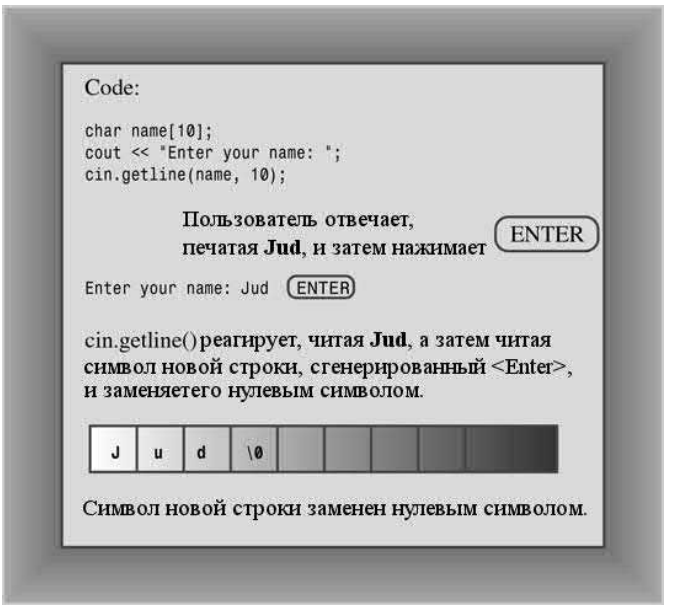


Рис. 4.5. *`getline()` читает и заменяет символ новой строки*

Строчно-ориентированный ввод с помощью `get()`

Теперь попробуем другой подход. Класс `istream` имеет функцию-член `get()`, которая доступна в различных вариантах. Один из них работает почти так же, как `getline()`. Он принимает те же аргументы, интерпретирует их аналогичным образом, и читает до конца строки. Но вместо того, чтобы прочитать и отбросить символ новой строки, `get()` оставляет его во входной очереди. Предположим, что используются два вызова `get()` подряд:

```
cin.get(name, ArSize);
cin.get(dessert, Arsize);     // проблема
```

Поскольку первый вызов оставляет символ новой строки во входной очереди, получается, что символ новой строки оказывается первым символом, который видит следующий вызов. Таким образом, второй вызов `get()` заключает, что он достиг конца строки, не найдя ничего интересного, что можно было бы прочитать. Без посторонней помощи `get()` вообще не может преодолеть этот символ новой строки.

К счастью, на помощь приходят различные варианты `get()`. Вызов `cin.get()` без аргументов читает одиночный следующий символ, даже если им будет символ новой строки, поэтому вы можете использовать его для того, чтобы отбросить символ новой строки и подготовиться к вводу следующей строки. То есть следующая последовательность будет работать правильно:

```
cin.get(name, ArSize);       // чтение первой строки
cin.get();                   // чтение символа новой строки
cin.get(dessert, Arsize);    // чтение второй строки
```

Другой способ применения get() состоит в *конкатенации*, или соединении, двух вызовов функций-членов класса, как показано в следующем примере:

```
cin.get(name, ArSize).get();     // конкатенация функций-членов
```

Такую возможность обеспечивает то, что cin.get(name, ArSize) возвращает объект cin, который затем используется в качестве объекта, вызывающего функцию get(). Аналогично приведенный ниже оператор читает две следующих друг за другом строки в массивы name1 и name2, что эквивалентно двум отдельным вызовам cin.getline():

```
cin.getline(name1, ArSize).getline(name2, ArSize);
```

В листинге 4.5 применяется конкатенация. В главе 11 вы узнаете о том, как включить это средство в собственные определения классов.

Листинг 4.5. insrt3.cpp

```
// insrt3.cpp -- чтение более одного слова с помощью get() и get()
#include <iostream>
int main()
{
    using namespace std;
    const int ArSize = 20;
    char name[ArSize];
    char dessert[ArSize];

    cout << "Enter your name:\n";              // запрос имени
    cin.get(name, ArSize).get();               // читать строку и символ новой строки
    cout << "Enter your favorite dessert:\n"; // запрос любимого десерта
    cin.get(dessert, ArSize).get();
    cout << "I have some delicious " << dessert;
    cout << " for you, " << name << ".\n";
    return 0;
}
```

Вот пример запуска программы из листинга 4.5:

```
Enter your name:
Mai Parfait
Enter your favorite dessert:
Chocolate Mousse
I have some delicious Chocolate Mousse for you, Mai Parfait.
```

Обратите внимание на то, что C++ допускает существование множества версий функции с разными списками аргументов.

Если вы используете, скажем, cin.get(name, ArSize), то компилятор определяет, что вызывается форма, которая помещает строку в массив, и подставляет соответствующую функцию-член. Если же вместо этого вы применяете cin.get(), то компилятор видит, что вам нужна форма, которая читает один символ. В главе 8 это средство, называемое *перегрузкой функций*, рассматривается более подробно.

Зачем вообще может понадобиться вызывать get() вместо getline()? Во-первых, старые реализации могут не располагать getline(). Во-вторых, get() позволяет проявлять бо́льшую осторожность. Предположим, например, что вы воспользовались get(), чтобы прочитать строку в массив. Как вы определите, была прочитана полная строка или же чтение прервалось в связи с заполнением массива? Для этого нужно посмотреть на следующий в очереди символ. Если это символ новой строки,

значит, была прочитана вся строка. В противном случае строка была прочитана не полностью и еще есть что читать. Этот прием исследуется в главе 17. Короче говоря, `getline()` немного проще в применении, но `get()` упрощает проверку ошибок. Вы можете использовать любую из этих функций для чтения ввода; просто учитывайте различия в их поведении.

Пустые строки и другие проблемы

Что происходит после того, как функции `getline()` и `get()` прочитали пустую строку? Изначально предполагалось, что следующий оператор ввода должен получить указание, где завершил работу предыдущий вызов `getline()` или `get()`. Однако современная практика заключается в том, что после того, как функция `get()` (но не `getline()`) прочитает пустую строку, она устанавливает флажок, который называется `failbit`. Влияние этого флажка состоит в том, что последующий ввод блокируется, но вы можете восстановить его следующей командой:

```
cin.clear();
```

Другая потенциальная проблема связана с тем, что входная строка может быть длиннее, чем выделенное для нее пространство. Если входная строка длиннее, чем указанное количество символов, то и `getline()`, и `get()` оставляют избыточные символы во входной очереди. Однако `getline()` дополнительно устанавливает `failbit` и отключает последующий ввод.

В главах 5, 6 и 17 эти свойства и способы программирования с их учетом рассматриваются более подробно.

Смешивание строкового и числового ввода

Смешивание числового и строкового ввода может приводить к проблемам. Рассмотрим пример простой программы в листинге 4.6.

Листинг 4.6. `numstr.cpp`

```
// numstr.cpp -- строковый ввод после числового
#include <iostream>
int main()
{
   using namespace std;
   cout << "What year was your house built?\n";    // ввод года постройки дома
   int year;
   cin >> year;
   cout << "What is its street address?\n";        // ввод адреса
   char address[80];
   cin.getline(address, 80);
   cout << "Year built: " << year << endl;         // вывод года постройки
   cout << "Address: " << address << endl;         // вывод адреса
   cout << "Done!\n";
   return 0;
}
```

В результате выполнения программы из листинга 4.6 получаем следующий вывод:

```
What year was your house built?
1966
What is its street address?
Year built: 1966
Address:
Done!
```

Вы так и не получили возможности ввести адрес. Проблема в том, что когда cin читает год, то оставляет символ новой строки, сгенерированный нажатием <Enter>, во входной очереди. Затем cin.getline() читает символ новой строки просто как пустую строку, после чего присваивает массиву address нулевую строку. Чтобы исправить это, нужно перед чтением адреса прочитать и отбросить символ новой строки. Это может быть сделано несколькими способами, включая вызов get() без аргументов либо с аргументом char, как описано в предыдущем примере. Эти вызовы можно выполнить по отдельности:

```
cin >> year;
cin.get(); // или cin.get(ch);
```

Или же можно сцепить вызов, воспользовавшись тем фактом, что выражение cin >> year возвращает объект cin:

```
(cin >> year).get(); // или (cin >> year).get(ch);
```

Если внести одно из таких исправлений в листинг 4.6, программа станет работать правильно:

```
What year was your house built?
1966
What is its street address?
43821 Unsigned Short Street
Year built: 1966
Address: 43821 Unsigned Short Street
Done!
```

Для обработки строк в программах на C++ часто используются указатели вместо массивов. Мы обратимся к этому аспекту строк после того, как немного поговорим об указателях. А пока рассмотрим более современный способ обработки строк: класс C++ по имени string.

Введение в класс string

В стандарте ISO/ANSI C++98 библиотека C++ была расширена за счет добавления класса string. Поэтому отныне вместо использования символьных массивов для хранения строк можно применять переменные типа string (или, пользуясь терминологией C++, объекты). Как вы увидите, класс string проще в использовании, чем массив, и к тому же предлагает более естественное представление строки как типа.

Для работы с классом string в программе должен быть включен заголовочный файл string. Класс string является частью пространства имен std, поэтому вы должны указать директиву using или объявление либо же ссылаться на класс как std::string. Определение класса скрывает природу строки как массива символов и позволяет трактовать ее как обычную переменную. В листинге 4.7 проиллюстрированы некоторые сходства и различия между объектами string и символьными массивами.

Листинг 4.7. strtype1.cpp

```
// strtype1.cpp -- использование класса C++ string
#include <iostream>
#include <string> // обеспечение доступа к классу string
int main()
{
   using namespace std;
```

```
    char charr1[20];                    // создание пустого массива
    char charr2[20] = "jaguar";         // создание инициализированного массива
    string str1;                        // создание пустого объекта строки
    string str2 = "panther";            // создание инициализированного объекта строки

    cout << "Enter a kind of feline: ";
         // Введите животное из семейства кошачьих
    cin >> charr1;
    cout << "Enter another kind of feline: ";
         // Введите другое животное из семейства кошачьих
    cin >> str1;                        // использование cin для ввода
    cout << "Here are some felines:\n";
    cout << charr1 << " " << charr2 << " "
        << str1 << " " << str2          // использование cout для вывода
        << endl;
    cout << "The third letter in " << charr2 << " is "
        << charr2[2] << endl;
    cout << "The third letter in " << str2 << " is "
        << str2[2] << endl;             // использование нотации массивов
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.7:

```
Enter a kind of feline: ocelot
Enter another kind of feline: tiger
Here are some felines:
ocelot jaguar tiger panther
The third letter in jaguar is g
The third letter in panther is n
```

Из этого примера вы должны сделать вывод, что во многих отношениях объект `string` можно использовать так же, как символьный массив.

- Объект `string` можно инициализировать строкой в стиле C.
- Чтобы сохранить клавиатурный ввод в объекте `string`, можно использовать `cin`.
- Для отображения объекта `string` можно применять `cout`.
- Можно использовать нотацию массивов для доступа к индивидуальным символам, хранящимся в объекте `string`.

Главное отличие между объектами `string` и символьными массивами, продемонстрированное в листинге 4.7, заключается в том, что объект `string` объявляется как обычная переменная, а не массив:

```
string str1;                // создание пустого объекта строки
string str2 = "panther";    // создание инициализированного объекта строки
```

Проектное решение, положенное в основу класса, позволяет программе автоматически обрабатывать изменение размера строк. Например, объявление `str1` создает объект `string` нулевой длины, но при чтении ввода в `str1` программа автоматически его увеличивает:

```
cin >> str1;                // str1 увеличен для того, чтобы вместить ввод
```

Это делает использование объекта `string` более удобным и безопасным по сравнению с массивом. Концептуально важным является то, что массив строк — это кол-

лекция единиц хранения отдельных символов, служащих для сохранения строки, а класс string – единая сущность, представляющая строку.

Инициализация строк в C++11

Как и можно было ожидать, C++11 позволяет осуществлять списковую инициализацию для строк в стиле C и объектов string:

```
char first_date[] = {"Le Chapon Dodu"};
char second_date[] {"The Elegant Plate"};
string third_date = {"The Bread Bowl"};
string fourth_date {"Hank's Fine Eats"};
```

Присваивание, конкатенация и добавление

Некоторые операции со строками класс string выполняет проще, чем это возможно в случае символьных массивов. Например, просто присвоить один массив другому нельзя. Однако один объект string вполне можно присвоить другому:

```
char charr1[20];          // создание пустого массива
char charr2[20] = "jaguar";  // создание инициализированного массива
string str1;              // создание пустого объекта string
string str2 = "panther"; // создание инициализированной строки
charr1 = charr2;          // НЕ ПРАВИЛЬНО, присваивание массивов не разрешено
str1 = str2;              // ПРАВИЛЬНО, присваивание объектов допускается
```

Класс string упрощает комбинирование строк. С помощью операции + можно сложить два объекта string вместе, а посредством операции += можно добавить строку к существующему объекту string. В отношении предшествующего кода у нас есть следующие возможности:

```
string str3;
str3 = str1 + str2;       // присвоить str3 объединение строк
str1 += str2;             // добавить str2 в конец str1
```

В листинге 4.8 приведен соответствующий пример. Обратите внимание, что складывать и добавлять к объектам string можно как другие объекты string, так и строки в стиле C.

Листинг 4.8. strtype2.cpp

```
// strtype2.cpp -- присваивание, сложение, добавление
#include <iostream>
#include <string>              // обеспечение доступа к классу string
int main()
{
    using namespace std;
    string s1 = "penguin";
    string s2, s3;

    // Присваивание одного объекта string другому
    cout << "You can assign one string object to another: s2 = s1\n";
    s2 = s1;
    cout << "s1: " << s1 << ", s2: " << s2 << endl;

    // Присваивание строки в стиле C объекту string
    cout << "You can assign a C-style string to a string object.\n";
    cout << "s2 = \"buzzard\"\n";
    s2 = "buzzard";
    cout << "s2: " << s2 << endl;
```

```
/
   // Конкатенация строк
   cout << "You can concatenate strings: s3 = s1 + s2\n";
   s3 = s1 + s2;
   cout << "s3: " << s3 << endl;

   // Добавление строки
   cout << "You can append strings.\n";
   s1 += s2;
   cout <<"s1 += s2 yields s1 = " << s1 << endl;
   s2 += " for a day";
   cout <<"s2 += \" for a day\" yields s2 = " << s2 << endl;
   return 0;
}
```

Вспомните, что управляющая последовательность \" представляет двойную кавычку, используемую как литеральный символ, а не ограничитель строки. Ниже показан вывод программы из листинга 4.8:

```
You can assign one string object to another: s2 = s1
s1: penguin, s2: penguin
You can assign a C-style string to a string object.
s2 = "buzzard"
s2: buzzard
You can concatenate strings: s3 = s1 + s2
s3: penguinbuzzard
You can append strings.
s1 += s2 yields s1 = penguinbuzzard
s2 += " for a day" yields s2 = buzzard for a day
```

Дополнительные сведения об операциях класса `string`

Еще до появления в C++ класса `string` программисты нуждались в таких действиях, как присваивание строк. Для строк в стиле C использовались функции из стандартной библиотеки C. Эти функции поддерживаются заголовочным файлом `cstring` (бывший `string.h`). Например, вы можете применять функцию `strcpy()` для копирования строки в символьный массив, а функцию `strcat()` — для добавления строки к символьному массиву:

```
strcpy(charr1, charr2);  // копировать charr2 в charr1
strcat(charr1, charr2);  // добавить содержимое charr2 к charr1
```

В листинге 4.9 сравниваются подходы с объектами `string` и символьными массивами.

Листинг 4.9. `strtype3.cpp`

```
// strtype3.cpp -- дополнительные средства класса string
#include <iostream>
#include <string>              // обеспечение доступа к классу string
#include <cstring>             // библиотека обработки строк в стиле C
int main()
{
   using namespace std;
   char charr1[20];
   char charr2[20] = "jaguar";
   string str1;
   string str2 = "panther";
```

```
    // Присваивание объектов string и символьных массивов
    str1 = str2;                        // копирование str2 в str2
    strcpy(charr1, charr2);             // копирование charr2 в charr1

    // Добавление объектов string и символьных массивов
    str1 += " paste";                   // добавление " paste" в конец str1
    strcat(charr1, " juice");           // добавление " juice" в конец charr1

    // Определение длины объекта string и строки в стиле C
    int len1 = str1.size();             // получение длины str1
    int len2 = strlen(charr1);          // получение длины charr1
    cout << "The string " << str1 << " contains "
         << len1 << " characters.\n";
    cout << "The string " << charr1 << " contains "
         << len2 << " characters.\n";
    return 0;
}
```

Ниже представлен вывод программы из листинга 4.9:

```
The string panther paste contains 13 characters.
The string jaguar juice contains 12 characters.
```

Синтаксис работы с объектами `string` выглядит проще, чем использование строковых функций C. Это особенно проявляется при более сложных операциях. Например, эквивалент из библиотеки C следующего оператора:

```
str3 = str1 + str2;
```

будет таким:

```
strcpy(charr3, charr1);
strcat(charr3, charr2);
```

Более того, при работе с массивами всегда существует опасность, что целевой массив окажется слишком малым для того, чтобы вместить всю информацию.

Например:

```
char site[10] = "house";
strcat(site, " of pancakes");       // проблема с нехваткой памяти
```

Функция `strcat()` пытается скопировать все 12 символов в массив `site`, таким образом, переполняя выделенную память. Это может вызвать аварийное завершение программы, или же программа продолжит работать, но с поврежденными данными. Класс `string`, с его автоматическим расширением при необходимости, позволяет избегать проблем подобного рода. Библиотека C предлагает функции, подобные `strcat()` и `strcpy()`, которые называются `strncat()` и `strncpy()`. Эти функции работают более безопасно, принимая третий параметр, который задает максимально допустимый размер целевого массива, но их применение усложняет написание программ.

Обратите внимание, что для получения количества символов в строке используется разный синтаксис:

```
int len1 = str1.size();             // получение длины str1
int len2 = strlen(charr1);          // получение длины charr1
```

`strlen()` — это стандартная функция, которая принимает в качестве аргумента строку в стиле C и возвращает количество символов в ней. Функция `size()` обычно делает то же самое, но синтаксис ее вызова отличается. Вместо передачи аргумента

ее имени предшествует имя объекта str1, отделенное точкой. Как вы уже видели на примере метода put() в главе 3, этот синтаксис означает, что str1 — это объект, а size() — метод класса. Метод — это функция, которая может быть вызвана только объектом, принадлежащим классу, в котором определен данный метод. В данном конкретном случае str1 — объект string, а size() — метод класса string. Короче говоря, функции C используют аргументы для идентификации требуемой строки, а объект класса C++ string использует имя объекта и операцию точки для указания того, какую именно строку нужно взять.

Дополнительные сведения о вводе-выводе класса `string`

Как вы уже видели, можно использовать cin с операцией >> для чтения объекта string и cout с операцией << — для отображения объекта string, причем с тем же синтаксисом, что и в случае строк в стиле C. Однако чтение за один раз целой строки с пробелами вместо отдельного слова требует другого синтаксиса. В листинге 4.10 демонстрируется это отличие.

Листинг 4.10. `strtype4.cpp`

```
// strtype4.cpp -- ввод строки с пробелами
#include <iostream>
#include <string>                              // обеспечение доступа к классу string
#include <cstring>                             // библиотека обработки строк в стиле C
int main()
{
   using namespace std;
   char charr[20];
   string str;

   // Длина строки в charr перед вводом
   cout << "Length of string in charr before input: "
        << strlen(charr) << endl;
   // Длина строки в str перед вводом
   cout << "Length of string in str before input: "
        << str.size() << endl;
   cout << "Enter a line of text:\n";          // ввод строки текста
   cin.getline(charr, 20);                     // указание максимальной длины
   cout << "You entered: " << charr << endl;
   cout << "Enter another line of text:\n";   // ввод другой строки текста
   getline(cin, str);    // теперь cin - аргумент; спецификатор длины отсутствует
   cout << "You entered: " << str << endl;
   // Длина строки в charr после ввода
   cout << "Length of string in charr after input: "
        << strlen(charr) << endl;
   // Длина строки в str после ввода
   cout << "Length of string in str after input: "
        << str.size() << endl;
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.10:

```
Length of string in charr before input: 27
Length of string in str before input: 0
Enter a line of text:
peanut butter
You entered: peanut butter
```

```
Enter another line of text:
blueberry jam
You entered: blueberry jam
Length of string in charr after input: 13
Length of string in str after input: 13
```

Обратите внимание, что программа сообщает длину строки в массиве `charr` перед вводом как равную 27, т.е. больше, чем размер массива! Здесь происходят две вещи. Первая — содержимое неинициализированного массива не определено. Вторая — функция `strlen()` работает, просматривая массив, начиная с первого элемента, и подсчитывает количество байт до тех пор, пока не встретит нулевой символ. В этом случае первый нулевой символ встретился через несколько байт за пределами массива. Где именно встретится нулевой символ в неинициализированном массиве, определяется случаем, поэтому весьма вероятно, что при запуске этой программы вы получите другое значение.

Также отметьте, что длина строки `str` перед вводом равна 0. Это объясняется тем, что размер неинициализированного объекта `string` автоматически устанавливается в 0.

Следующий код читает строку в массив:

```
cin.getline(charr, 20);
```

Точечная нотация указывает на то, что функция `getline()` — это метод класса `istream`. (Вспомните, что `cin` является объектом класса `istream`.) Как упоминалось ранее, первый аргумент задает целевой массив, а второй — его размер, используемый `getline()` для того, чтобы избежать переполнения массива.

Следующий код читает строку в объект `string`:

```
getline(cin,str);
```

Здесь точечная нотация не используется, а это говорит о том, что *данная* функция `getline()` *не* является методом класса. Поэтому она принимает объект `cin` как аргумент, сообщающий о том, где искать ввод. К тому же нет аргумента, задающего размер строки, потому что объект `string` автоматически изменяет свой размер, чтобы вместить строку.

Так почему же одна функция `getline()` — метод класса `istream`, а вторая — нет? Класс `istream` появился в C++ до того, как был добавлен класс `string`. Поэтому `istream` распознает базовые типы C++, такие как `double` или `int`, но ничего не знает о типе `string`. Таким образом, класс `istream` имеет методы, обрабатывающие `double`, `int` и другие базовые типы, но не имеет методов, обрабатывающих объекты `string`.

Поскольку у класса `istream` нет методов, обрабатывающих объекты `string`, вас может удивить, почему работает следующий код:

```
cin >> str;     // чтение слова в объект string по имени str
```

Оказывается, что следующий код использует (в скрытом виде) функцию-член класса `istream`:

```
cin >> x;       // чтение значения в переменную базового типа C++
```

Но эквивалент этого кода с классом `string` использует дружественную функцию (также в скрытом виде) класса `string`. Вам придется подождать до главы 11 объяснений, что собой представляет дружественная функция и как этот прием работает. Между тем можете смело использовать `cin` и `cout` с объектами `string`, не заботясь о его внутреннем функционировании.

Другие формы строковых литералов

Вспомните, что в дополнение к char язык C++ имеет тип wchar_t. Также в C++11 были добавлены типы char16_t и char32_t. На основе этих типов можно создавать массивы и строковые литералы. Для строковых литералов перечисленных типов в C++ используются префиксы L, u и U соответственно. Ниже приведен пример:

```
wchar_t title[] = L"Chief Astrogator";      // строка w_char
char16_t name[] = u"Felonia Ripova";        // строка char_16
char32_t car[] = U"Humber Super Snipe";     // строка char_32
```

C++11 также поддерживает схему кодирования для символов Unicode под названием UTF-8. В зависимости от числового значения, в этой схеме символ может храниться в пределах от 8-битной единице, или октете, до четырех 8-битных единиц. Для указания строковых литералов такого типа в C++ используется префикс u8.

Другим добавлением C++11 является необработанная (raw) строка. В такой строке символы представляют сами себя. Например, последовательность \n не интерпретируется как представление символа новой строки; вместо этого она будет выглядеть как два обычных символа, обратная косая черта и n, отображаясь таким же образом на экране. Или еще один пример: внутри необработанной строки можно использовать просто двойную кавычку ", а не конструкцию \", как это было в листинге 4.8. Естественно, поскольку двойные кавычки можно применять внутри строкового литерала, их больше нельзя использовать для обозначения начала и конца строки. Таким образом, в необработанных строках в качестве разделителей применяются последовательности "(и)", а также префикс R для идентификации их как необработанных строк:

```
cout << R"(Jim "King" Tutt uses "\n" instead of endl.)" << '\n';
```

Этот оператор приводит к отображению следующей строки:

```
Jim "King" Tutt uses \n instead of endl.
```

Эквивалентный стандартный строковый литерал выглядит так:

```
cout << "Jim \"King\" Tutt uses \" \\n\" instead of endl." << '\n';
```

Здесь мы использовали \\ для отображения \, поскольку одиночный символ \ интерпретируется как первый символ управляющей последовательности.

Нажатие клавиши <Enter> или <Return> во время набора необработанной строки приводит не только к переходу курсора к следующей строке на экране, но также и помещению символа возврата каретки в саму необработанную строку.

А что если необходимо отобразить комбинацию)" в необработанной строке? Не будет ли компилятор интерпретировать первое вхождение)" как конец строки? Да, будет. Однако синтаксис необработанных строк позволяет помещать дополнительные символы между открывающими " и (. Это означает, что те же самые дополнительные символы должны присутствовать между закрывающими) и ". Таким образом, необработанная строка, начинающаяся с R"+* должна завешаться с помощью)+*". То есть оператор

```
cout << R"+*("(Who wouldn't?)", she whispered.)+*" << endl;
```

отобразит следующее:

```
"(Who wouldn't?)", she whispered.
```

Если кратко, то здесь разделители по умолчанию "(и)" были заменены последовательностями "+*(и)+*". В качестве части разделителя можно использовать

любые члены базового набора символов, за исключением пробела, открывающей и закрывающей круглых скобок, обратной косой черты и управляющих символов, таких как табуляция или новая строка.

Префикс `R` может быть скомбинирован с другими строковыми префиксами для получения необработанных строк из символов `wchar_t` и т.п. Этот префикс может быть первой или последней частью составного префикса: `Ru`, `UR` и т.д.

Теперь перейдем к рассмотрению другого составного типа — структуры.

Введение в структуры

Предположим, вы необходимо хранить информацию о баскетболисте. Вы хотите хранить его имя, зарплату, рост, вес, среднюю результативность, процент попаданий, результативных передач и т.п. Вам понадобится некоторая форма данных, которая могла бы хранить всю эту информацию как единое целое. Массив здесь не подойдет. Хотя массив может хранить несколько элементов, но все они должны быть одного типа. То есть один массив может хранить 20 целых чисел, другой — 10 чисел с плавающей точкой, однако массив не может хранить целые значения в одних элементах и значения с плавающей точкой — в других.

Удовлетворить вашу потребность в совместном хранении всей информации о баскетболисте может структура C++. *Структура* — более универсальная форма данных, нежели массив, потому что одна структура может хранить элементы более чем одного типа. Это позволяет унифицировать представление данных за счет сохранения всей информации, связанной с баскетболистом, в одной переменной типа структуры. Если вы хотите отслеживать информацию о целой команде, то можете воспользоваться массивом структур. Тип структуры — это еще и ступенька к покорению бастиона объектно-ориентированного программирования C++ — класса. Изучение структур приблизит вас к сердцу ООП на языке C++.

Структура представляет собой определяемый пользователем тип с объявлением, описывающим свойства данных типа. После определения типа можно создавать переменные этого типа. То есть создание структуры — процесс, состоящий из двух частей. Вначале определяется описание структуры, в котором перечисляются и именуются типы данных, хранящиеся в структуре. Затем создаются структурные переменные, или, иначе говоря, структурные объекты данных, которые следуют плану, заданному объявлением.

Например, предположим, что компания Bloataire, Inc. желает создать тип данных, описывающий линейку ее продуктов — различного рода надувных предметов. В частности, тип должен включать наименование продукта, его объем в кубических футах, а также розничную цену. Вот описание структуры, отвечающее этим потребностям:

```
struct inflatable   // объявление структуры
{
    char  name[20];
    float volume;
    ouble price;
};
```

Ключевое слово `struct` указывает на то, что этот код определяет план структуры. Идентификатор `inflatable` — имя, или *дескриптор*, этой формы, т.е. имя нового типа. Таким образом, теперь можно создавать переменные типа `inflatable` точно так же, как создаются переменные типа `char` или `int`. Далее между фигурными скобками находится список типов данных, которые будут содержаться в структуре. Каждый элемент списка — это оператор объявления. Здесь допускается использовать

любые типы C++, включая массивы и другие структуры. В этом примере применяется массив char, который подходит для хранения строки, затем идет один элемент типа float и один — типа double. Каждый индивидуальный элемент в списке называется *членом* структуры, так что структура inflatable имеет три члена (рис. 4.6). Выражаясь кратко, определение структуры описывает характеристики типа — в рассматриваемом случае типа inflatable.

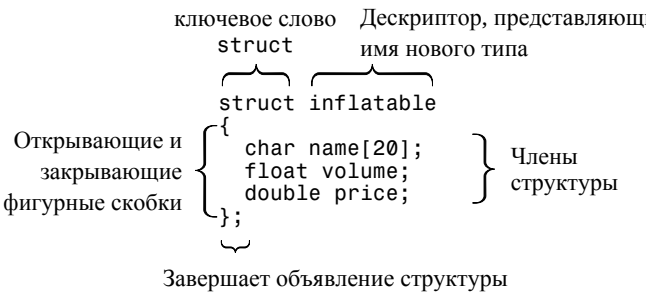


***Рис. 4.6.** Части описания структуры*

После определения структуры можно создавать переменные этого типа:

```
inflatable hat;              // hat — структурная переменная типа inflatable
inflatable woopie_cushion;   // переменная типа inflatable
inflatable mainframe;        // переменная типа inflatable
```

Если вы знакомы со структурами в языке C, то отметите (возможно, с удовольствием), что C++ позволяет отбросить ключевое слово struct при объявлении структурных переменных:

```
struct inflatable goose;     // ключевое слово struct требуется в C
inflatable vincent;          // ключевое слово struct не требуется в C++
```

В C++ дескриптор структуры используется в точности как имя фундаментального типа. Это изменение подчеркивает, что объявление структуры определяет новый тип. Кроме того, оно исключает пропуск слова struct из списка причин выдачи сообщений об ошибках компилятором.

При условии, что переменная hat имеет тип inflatable, для доступа к ее отдельным членам используется *операция принадлежности* или *членства* (.).

Например, hat.volume ссылается на член структуры по имени volume, а hat.price — на член по имени price. Аналогично, vincent.price — это член price переменной vincent. Короче говоря, имена членов позволяют ссылаться на члены структур почти так же, как индексы — на элементы массивов. Поскольку член price объявлен как double, члены hat.price и vincent.price являются эквивалентами переменных типа double и могут быть использованы точно так же, как любая другая переменная типа double. Короче говоря, hat является структурой, но hat.price относится к типу double. Кстати, метод, применяемый для доступа к функции-члену класса вроде cin.getline() происходит от метода доступа к переменным-членам структуры, таким как vincent.price.

Использование структур в программах

Теперь, когда раскрыты некоторые из основных свойств структур, пришло время собрать все идеи вместе в программе, использующей переменные-структуры.

В листинге 4.11 представлен пример со структурой. Также в нем показано, как ее инициализировать.

Листинг 4.11. structur.cpp

```
// structur.cpp -- простая структура
#include <iostream>
struct inflatable              // объявление структуры
{
   char name[20];
   float volume;
   double price;
};
int main()
{
   using namespace std;
   inflatable guest =
   {
      "Glorious Gloria",       // значение name
      1.88,                    // значение volume
      29.99                    // значение value
   }; // guest - структурная переменная типа inflatable
   // Инициализация указанными значениями
   inflatable pal =
   {
      "Audacious Arthur",
      3.12,
      32.99
   }; // pal - вторая переменная типа inflatable
   // ПРИМЕЧАНИЕ: некоторые реализации требуют использования
   // static inflatable guest =

   cout << "Expand your guest list with " << guest.name;
   cout << " and " << pal.name << "!\n";    // pal.name - член name переменной pal
   cout << "You can have both for $";
   cout << guest.price + pal.price << "!\n";
   return 0;
}
```

Ниже показан вывод программы из листинга 4.11:

```
Expand your guest list with Glorious Gloria and Audacious Arthur!
You can have both for $62.98!
```

Замечания по программе

Относительно программы в листинге 4.11 следует отметить один важный аспект – расположение объявления структуры. Для `structur.cpp` существует два варианта. Объявление можно разместить внутри функции `main()`. Второй вариант, использованный здесь, состоит в том, чтобы расположить объявление за пределами функции `main()`. Когда объявление встречается вне функции, оно называется *внешним объявлением*. Для конкретной программы нет практической разницы между этими двумя вариантами. Но для программ, состоящих из двух и более функций, разница может оказаться существенной. Внешнее объявление может быть использовано всеми функциями, которые следуют за ней, в то время как внутреннее объявление может применяться только той функцией, в которой это объявление находится. Чаще всего вам придется применять внешнее объявление, чтобы все функции могли работать со структурами этого типа (рис. 4.7).

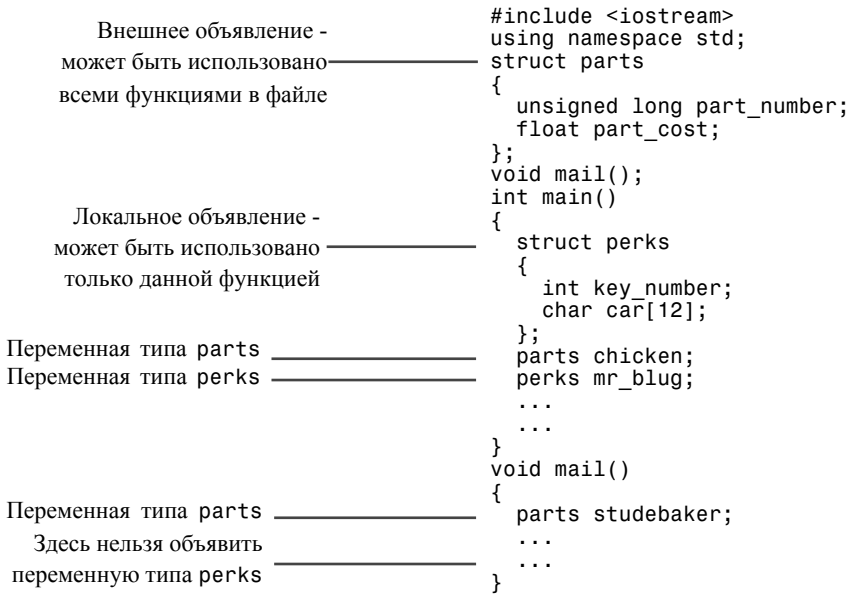


***Рис. 4.7.** Локальные и внешние объявления структур*

Переменные также могут быть определены как внутренние или внешние, причем внешние переменные доступны всем функциям. (В главе 9 эта тема рассматривается более подробно.) В C++ не поощряется использование внешних переменных, но приветствуется применение внешних объявлений структур. К тому же часто имеет смысл объявлять внешними и символические константы.

Теперь обратите внимание на процедуру инициализации:

```
inflatable guest =
{
   "Glorious Gloria",    // значение name
   1.88,                 // значение volume
   29.99                 // значение value
};
```

Как и в случае массивов, здесь используется список значений, разделенных запятыми, внутри пары фигурных скобок. В примере программы каждое значение находится в отдельной строке, но их все можно было бы поместить в одну строку. Главное — не забыть разделять их запятыми:

```
inflatable duck = {"Daphne", 0.12, 9.98};
```

Каждый член структуры можно инициализировать данными соответствующего вида. Например, член структуры `name` — это символьный массив, поэтому его можно инициализировать строкой. Каждый член структуры трактуется как переменная этого типа. То есть `pal.price` — переменная типа `double`, а `pal.name` — массив `char`. И когда программа использует `cout`, чтобы отобразить `pal.name`, она отображает этот член как строку. Кстати, поскольку `pal.name` — символьный массив, для доступа к его отдельным символам можно использовать индексы. Например, `pal.name[0]` содержит символ `A`. Однако `pal[0]`не имеет смысла, потому что `pal` является структурой, а не массивом.

Инициализация структур в C++11

Как и в случае массивов, C++11 расширяет возможности списковой инициализации. Знак = является необязательным:

```
inflatable duck {"Daphne", 0.12, 9.98};  // в C++11 знак = можно опустить
```

Пустые фигурные скобки приводят к установке индивидуальных членов в 0. Например, следующее объявление обеспечивает установку в 0 членов `mayor.volume` и `mayor.price`, а также всех байтов в `mayor.name`:

```
inflatable mayor {};
```

И, наконец, сужение не допускается.

Может ли структура содержать член типа `string`?

Можно ли использовать объект класса `string` вместо символьного массива в качестве типа члена `name`? То есть, допустимо ли следующее объявление структуры:

```
#include <string>
struct inflatable  // определение структуры
{
   std::string name;
   float volume;
   double price;
};
```

Ответ положительный при условии, что не используется устаревший компилятор, который не поддерживает инициализацию структур с помощью членов класса `string`.

Удостоверьтесь, что определение структуры имеет доступ к пространству имен `std`. Это можно обеспечить, поместив директиву `using` выше определения структуры. Более удачным решением, как было показано ранее, является объявление `name` с типом `std::string`.

Прочие свойства структур

В C++ пользовательские типы сделаны, насколько возможно, похожими на встроенные типы. Например, структуру можно передавать как аргумент функции, а функция может использовать структуру в качестве возвращаемого значения. Также можно применять операцию присваивания (=), чтобы присвоить одну структуру другой того же самого типа. Эта операция устанавливает значение каждого члена одной структуры равным значению соответствующего члена другой структуры, даже если член является массивом. Такой тип присваивания называется *почленным присваиванием*. Мы отложим разговор о передаче и возврате структур до обсуждения темы функций в главе 7, но приведем небольшой пример присваивания структур в листинге 4.12.

Листинг 4.12. `assign_st.cpp`

```
// assign_st.cpp -- присваивание структур
#include <iostream>
struct inflatable
{
   char name[20];
   float volume;
   double price;
};
```

```
int main()
{
   using namespace std;
   inflatable bouquet =
   {
      "sunflowers",
      0.20,
      12.49
   };
   inflatable choice;
   cout << "bouquet: " << bouquet.name << " for $";
   cout << bouquet.price << endl;
   choice = bouquet;          // присваивание одной структуры другой
   cout << "choice: " << choice.name << " for $";
   cout << choice.price << endl;
   return 0;
}
```

Вот как выглядит вывод программы из листинга 4.12:

```
bouquet: sunflowers for $12.49
choice: sunflowers for $12.49
```

Как видите, почленное присваивание работает, и все члены структуры `choice` получили соответствующие значения членов структуры `bouquet`.

Можно комбинировать определение формы структуры с созданием структурных переменных. Чтобы сделать это, сразу после закрывающей фигурной скобки понадобится указать имя переменной или нескольких переменных:

```
struct perks
{
   int key_number;
   char car[12];
} mr_smith, ms_jones;     // две переменных типа perks
```

Можно даже инициализировать созданную переменную, как показано ниже:

```
struct perks
{
   int key_number;
   char car[12];
} mr_glitz =
{
   7,                     // значение члена mr_glitz.key_number
   "Packard"              // значение члена mr_glitz.car
};
```

Однако отделение определения структуры от объявлений переменных обычно повышает читабельность программы.

Еще один трюк, который можно сделать со структурой — создать структуру без имени типа. При определении имя дескриптора опускается и сразу следует имя переменной:

```
struct                    // дескриптора нет
{
   int x;                 // два члена
   int y;
} position;               // структурная переменная
```

Это создает одну структурную переменную по имени position. К ее членам можно обращаться через операцию точки, как в position.x, но никакого общего имени для типа не объявляется. Вы не сможете впоследствии создавать другие переменные того же типа. В настоящей книге мы эта ограниченная форма структур использоваться не будет.

Помимо того факта, что программа C++ может использовать дескриптор структуры в качестве имени типа, все остальные характеристики структур присущи как структурам C, так и структурам C++, не считая изменений, появившихся в C++11. Однако структуры C++ двигаются еще дальше. В отличие от структур C, например, структуры C++ могут включать в себя функции-члены в дополнение к переменным-членам. Однако эти более развитые средства чаще используются с классами, чем со структурами, поэтому мы поговорим о них, когда раскроем тему классов — в главе 10.

Массивы структур

Структура inflatable содержит массив (по имени name). Также можно создавать массивы, элементами которых являются структуры. Подход здесь в точности совпадает с таковым для массивов фундаментальных типов. Например, чтобы создать массив из 100 структур inflatable, можно поступить так:

```
inflatable gifts[100];          // массив из 100 структур inflatable
```

Это объявление делает gifts массивом структур inflatable. В результате каждый элемент массива, такой как gifts[0] или gifts[99], является объектом типа inflatable и может быть использован с операцией членства:

```
cin >> gifts[0].volume;          // используется член volume первой структуры
cout << gifts[99].price << endl; // отображается член price последней структуры
```

Имейте в виду, что сам по себе gifts является массивом, а не структурой, поэтому конструкция вроде gifts.price — некорректна.

Для инициализации массива структур комбинируется правило инициализации массивов (заключенный в фигурные скобки список значений, разделенных запятыми, для каждого элемента) с правилом структур (заключенный в фигурные скобки список значений, разделенных запятыми, для каждого члена).

Поскольку каждый элемент массива является структурой, его значение представляется инициализацией структуры. Таким образом, мы получаем следующую конструкцию:

```
inflatable guests[2] =          // инициализация массива структур
{
    {"Bambi", 0.5, 21.99},      // первая структура в массиве
    {"Godzilla", 2000, 565.99}  // следующая структура в массиве
};
```

Как обычно, можете сформатировать все это по своему усмотрению. Например, обе инициализации могут быть расположены в одной строке или же инициализация каждого отдельного члена структуры может занимать отдельную строку.

В листинге 4.13 показан пример использования массива структур. Обратите внимание, что поскольку guests — массив inflatable, типом элемента guests[0] является inflatable, поэтому вы можете использовать его с операцией точки для доступа к членам структуры inflatable.

Листинг 4.13. arrstruc.cpp

```
// arrstruc.cpp -- массив структур
#include <iostream>
struct inflatable
{
   char   name[20];
   float  volume;
   double price;
};
int main()
{
   using namespace std;
   inflatable guests[2] =          // инициализация массива структур
   {
      {"Bambi", 0.5, 21.99},       // первая структура в массиве
      {"Godzilla", 2000, 565.99}   // следующая структура в массиве
   };
   cout << "The guests " << guests[0].name << " and " << guests[1].name
        << "\nhave a combined volume of "
        << guests[0].volume + guests[1].volume << " cubic feet.\n";
   return 0;
}
```

Ниже показан вывод программы из листинга 4.13:

```
The guests Bambi and Godzilla
have a combined volume of 2000.5 cubic feet.
```

Битовые поля в структурах

Язык C++, как и C, позволяет указывать члены структур, занимающие определенное количество битов памяти. Это может пригодиться для создания структур данных, которые соответствуют, скажем, регистру в некотором аппаратном устройстве. Тип поля должен быть целочисленным или перечислимым (перечисления обсуждаются далее в этой главе) и после него вслед за двоеточием ставится число, указывающее действительное количество битов. Для выравнивания можно применять неименованные поля. Каждый член называется *битовым полем*. Вот пример:

```
struct torgle_register
{
   unsigned int SN : 4;        // 4 бита для значения SN
   unsigned int : 4;           // 4 бита не используются
   bool goodIn : 1;            // допустимый ввод (1 бит)
   bool goodTorgle : 1;        // признак успешности
};
```

Эти поля можно инициализировать в обычной манере и применять стандартную нотацию структур для доступа к ним:

```
torgle_register tr = { 14, true, false };
...
if (tr.goodIn)                 // оператор if описан в главе 6
...
```

Битовые поля обычно применяются в низкоуровневом программировании. Альтернативным подходом является использование целочисленного типа и битовых операций, перечисленных в приложении Д.

Объединения

Объединение — это формат данных, который может хранить в пределах одной области памяти разные типы данных, но в каждый момент времени только один из них. То есть, в то время как структура может содержать, скажем, int, long и double, объединение может хранить либо int, либо long, либо double. Синтаксис похож на синтаксис структур, но смысл отличается. Например, рассмотрим следующее объявление:

```
union one4all
{
   int int_val;
   long long_val;
   double double_val;
};
```

Переменную one4all можно использовать для хранения int, long или double, если только делать это не одновременно:

```
one4all pail;
pail.int_val = 15;              // сохранение int
cout << pail.int_val;
pail.double_val = 1.38;         // сохранение double, int теряется
cout << pail.double_val;
```

Таким образом, pail может служить в качестве переменной int в одном случае и в качестве переменной double — в другом. Имя члена идентифицирует роль, в которой в данный момент выступает переменная. Поскольку объединение хранит только одно значение в единицу времени, оно должно иметь достаточный размер, чтобы вместить самый большой член. Поэтому размер объединения определяется размером его самого большого члена.

Причиной применений объединения может быть необходимость сэкономить память, когда элемент данных может использовать два или более форматов, но никогда — одновременно. Например, предположим, что вы ведете реестр каких-то предметов, из которых одни имеют целочисленный идентификатор, а другие — строковый. В этом случае можно применить следующий подход:

```
struct widget
{
   char brand[20];
   int type;
   union id                     // формат зависит от типа предмета
   {
      long id_num;              // предметы первого типа
      char id_char[20];         // прочие предметы
   } id_val;
};
...
widget prize;
...
if (prize.type == 1)            // оператор if-else (глава 6)
   cin >> prize.id_val.id_num; // использование поля name для указания режима
else
   cin >> prize.id_val.id_char;
```

Анонимное объединение не имеет имени; в сущности, его члены становятся переменными, расположенными по одному и тому же адресу в памяти. Разумеется, только одна из них может быть текущей в единицу времени:

```
struct widget
{
   char brand[20];
   int type;
   union                        // анонимное объединение
   {
      long id_num;              // предметы первого типа
      char id_char[20];         // прочие предметы
   };
};
...
widget prize;
...
if (prize.type == 1)
   cin >> prize.id_num;
else
   cin >> prize.id_char;
```

Поскольку объединение является анонимным, `id_num` и `id_char` трактуются как два члена `prize`, разделяющие один и тот же адрес памяти. Необходимость в промежуточном идентификаторе `id_val` отпадает. То, какое поле активно в каждый момент времени, остается на усмотрение программиста.

Объединения часто (но не только) используются для экономии пространства памяти. В наши дни, когда доступны гигабайты ОЗУ и терабайты на жестких дисках, это может показаться не особенно важным, но не все программы на C++ ориентированы на системы подобного рода. Язык C++ также применяется для встроенных систем, таких как процессоры, управляющие духовым шкафом, MP3-проигрывателем или марсоходом. В таких приложениях пространство может быть дефицитным ресурсом. Кроме того, объединения часто используются при работе с операционными системами или аппаратными структурами данных.

Перечисления

Средство C++ `enum` представляет собой альтернативный по отношению к `const` способ создания символических констант. Он также позволяет определять новые типы, но в очень ограниченной манере. Синтаксис `enum` подобен синтаксису структур. Например, рассмотрим следующий оператор:

```
enum spectrum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

Этот оператор делает две вещи.

- Объявляет имя нового типа – `spectrum`; при этом `spectrum` называется *перечислением*, почти так же, как переменная `struct` называется структурой.
- Устанавливает `red`, `orange`, `yellow` и т.д. в качестве символических констант для целочисленных значений 0–7. Эти константы называются *перечислителями*.

По умолчанию перечислителям присваиваются целочисленные значения, начиная с 0 для первого из них, 1 – для второго и т.д. Это правило по умолчанию можно переопределить, явно присваивая целочисленные значения. Чуть позже вы увидите, как это делается.

Имя перечисления можно использовать для объявления переменной с этим типом перечисления:

```
spectrum band;              // band – переменная типа spectrum
```

Переменные типа перечислений имеют ряд специальных свойств, которые мы сейчас рассмотрим.

Единственными допустимыми значениями, которые можно присвоить переменной типа перечисления без необходимости приведения типов, являются значения, указанные в определении этого перечисления. Рассмотрим пример:

```
band = blue;                // правильно, blue - перечислитель
band = 2000;                // неправильно, 2000 – не перечислитель
```

Таким образом, переменная `spectrum` ограничена только восемью допустимыми значениями. Некоторые компиляторы выдают ошибку, если вы пытаетесь присвоить некорректное значение, в то время как другие выдают только предупреждения. Для максимальной переносимости вы должны трактовать присваивание переменным типа `enum` значений, не входящих в определение `enum`, как ошибку.

Для перечислений определена только операция присваивания. В частности, арифметические операции не предусмотрены:

```
band = orange;              // правильно
++band;                     // неправильно (операция ++ обсуждается в главе 5)
band = orange + red;        // неправильно, но довольно хитро
...
```

Однако некоторые реализации не накладывают таких ограничений. Это позволяет нарушить ограничения типа. Например, если `band` равно `ultraviolet`, или 7, а затем выполняется `++band`, и если такое разрешено компилятором, то `band` получит значение, недопустимое для типа `spectrum`. Опять-таки, для достижения максимальной переносимости вы должны придерживаться ограничений.

Перечисления – целочисленные типы, и они могут быть представлены в виде `int`, однако тип `int` не преобразуется автоматически в тип перечисления:

```
int color = blue;           // правильно, тип spectrum приводится к int
band = 3;                   // неправильно, int не преобразуется в spectrum
color = 3 + red;            // правильно, red преобразуется в int
...
```

Обратите внимание, что хотя значение `3` в этом примере соответствует перечислителю `green`, все же присваивание `3` переменной `band` вызывает ошибку несоответствия типа. Но присваивание `green` переменной `band` законно, потому что оба они имеют тип `spectrum`. И снова, некоторые реализации не накладывают такого ограничения. В выражении `3 + red` сложение не определено для перечислений. Однако `red` преобразуется в тип `int`, в результате чего получается значение типа `int`. Благодаря преобразованию перечисления в `int` в данной ситуации, вы можете использовать перечислители в арифметических выражениях, комбинируя их с обыч- ными целыми, даже несмотря на то, что такая арифметика не определена для самих перечислителей.

Предыдущий пример

```
band = orange + red;        // неправильно, но довольно хитро
```

не работает по другой причине. Да, действительно, операция + не определена для перечислителей. Но также верно и то, что перечислители преобразуются в целые числа, когда применяются в арифметических выражениях, поэтому выражение

orange + red превращается в 1 + 0, что вполне корректно. Но это выражение имеет тип int, поэтому оно не может быть присвоено переменной band типа spectrum.

Вы можете присвоить значение int переменной enum, если полученное значение допустимо и применяется явное приведение типа:

```
band = spectrum(3);          // приведение 3 к типу spectrum
```

Но что будет, если вы попытаетесь выполнить приведение типа для недопустимого значения? Результат не определен, в том смысле, что попытка не будет воспринята как ошибочная, но вы не можете полагаться на полученное в результате значение:

```
band = spectrum(40003);      // результат не определен
```

(Обсуждение того, какие значения являются приемлемыми, а какие неприемлемыми, ищите в разделе "Диапазоны значений перечислителей" далее в этой главе.)

Как видите, правила, которым подчиняются перечисления, достаточно строги. На практике перечисления чаще используются как способ определения взаимосвязанных символических констант, нежели как средство определения новых типов. Например, вы можете применять перечисления для определения символических констант для операторов switch. (Примеры ищите в главе 6.) Если вы собираетесь только использовать константы и не создавать переменные перечислимого типа, то в этом случае можете опустить имя перечислимого типа, как показано ниже:

```
enum {red, orange, yellow, green, blue, violet, indigo, ultraviolet};
```

Установка значений перечислителей

Конкретные значения элементов перечислений можно устанавливать явно посредством операция присваивания:

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
```

Присваиваемые значения должны быть целочисленными. Можно также явно устанавливать только некоторые из перечислителей:

```
enum bigstep{first, second = 100, third};
```

В этом случае first получает значение 0 по умолчанию. Каждый последующий неинициализированный перечислитель увеличивается на единицу по сравнению с предыдущим. Поэтому third имеет значение 101.

И, наконец, допускается указывать одно и то же значение для нескольких перечислителей:

```
enum {zero, null = 0, one, numero_uno = 1};
```

Здесь zero и null имеют значение 0, а one и numero_uno — значение 1. В ранних версиях C++ элементам перечислений можно было присваивать только значения типа int (или неявно преобразуемые к int), но теперь это ограничение снято, и также можно использовать значения типа long или даже long long.

Диапазоны значений перечислителей

Изначально правильными значениями перечислений являются лишь те, что названы в объявлении. Однако C++ расширяет список допустимых значений, которые могут быть присвоены перечислимым переменным, за счет использования приведения типа. Каждое перечисление имеет *диапазон*, и с помощью приведения к типу переменной перечисления можно присвоить любое целочисленное значение в пределах этого диапазона, даже если данное значение не равно ни одному из перечислителей.

Например, предположим, что bits и myflag определены следующим образом:

```
enum bits{one = 1, two = 2, four = 4, eight = 8};
bits myflag;
```

В таком случае показанный ниже оператор является допустимым:

```
myflag = bits(6);  // правильно, потому что 6 находится в пределах диапазона
```

Здесь 6 не является значением ни одного из перечислителей, однако находится в диапазоне этого определенного перечисления.

Диапазон определяется следующим образом. Для нахождения верхнего предела выбирается перечислитель с максимальным значением. Затем ищется наименьшее число, являющееся степенью двойки, которое больше этого максимального значения, и из него вычитается единица. (Например, максимальное значение bigstep, как определено выше, равно 101. Минимальное число, представляющее степень двойки, которое больше 101, равно 128, поэтому верхним пределом диапазона будет 127.) Для нахождения минимального предела выбирается минимальное значение перечислителя. Если оно равно 0 или больше, то нижним пределом диапазона будет 0. Если же минимальное значение перечислителя отрицательное, используется такой же подход, как при вычислении верхнего предела, но со знаком минус. (Например, если минимальный перечислитель равен -6, то следующей степенью двойки будет -8, и нижний предел получается равным -7.)

Идея состоит в том, чтобы компилятор мог выяснить, сколько места необходимо для хранения перечисления. Он может использовать от 1 байт или менее для перечислений с небольшим диапазоном, и до 4 байт — для перечислений со значениями типа long.

Стандарт C++11 расширяет перечисления добавлением формы, которая называется *ограниченным перечислением*. Эта форма кратко в главе 10.

Указатели и свободное хранилище

В начале главы 3 упоминалось о трех фундаментальных свойствах, которые должна отслеживать компьютерная программа, когда она сохраняет данные.

Напомним их:

- где хранится информация;
- какое значение сохранено;
- разновидность сохраненной информации.

Пока что вы использовали только одну стратегию: объявление простых переменных. В операторе объявления предоставляется тип и символическое имя значения. Он также заставляет программу выделить память для этого значения и внутренне отслеживать ее местоположение.

Давайте рассмотрим другую стратегию, важность которой проявляется при разработке классов C++. Эта стратегия основана на указателях, которые представляют собой переменные, хранящие адреса значений вместо самих значений. Но прежде чем обратиться к указателям, давайте поговорим о том, как явно получить адрес обычной переменной. Для этого применяется операция взятия адреса, обозначаемая символом &, к переменной, адрес которой интересует. Например, если home — переменная, то &home — ее адрес. В листинге 4.14 демонстрируется использование этой операции.

Листинг 4.14. address.cpp

```
// address.cpp -- использование операции & для нахождения адреса
#include <iostream>
int main()
{
   using namespace std;
   int donuts = 6;
   double cups = 4.5;

   cout << "donuts value = " << donuts;
   cout << " and donuts address = " << &donuts << endl;
   // ПРИМЕЧАНИЕ: может понадобиться использовать
   // unsigned (&donuts) и unsigned (&cups)
   cout << "cups value = " << cups;
   cout << " and cups address = " << &cups << endl;
   return 0;
}
```

Ниже показан вывод программы из листинга 4.14 в одной из систем:

```
donuts value = 6 and donuts address = 0x0065fd40
cups value = 4.5 and cups address = 0x0065fd44
```

В показанной здесь конкретной реализации `cout` используется шестнадцатеричная нотация при отображении значений адресов, т.к. это обычная нотация, применяемая для указания адресов памяти. (Некоторые реализации применяют десятичную нотацию.) Наша реализация сохраняет `donuts` в памяти с меньшими адресами, чем `cups`. Разница между этими двумя адресами составляет `0x0065fd44-0x0065fd40`, или 4 байта. Конечно, в разных системах вы получите разные значения этих адресов. К тому же некоторые системы могут сохранять `cups` перед `donuts`, и разница между адресами составит 8, потому что `cups` имеет тип `double`. Другие системы могут даже разместить эти переменные в памяти далеко друг от друга, а не рядом.

Таким образом, использование обычных переменных трактует значение как именованную величину, а ее местоположение — как производную величину. Теперь рассмотрим стратегию указателей, которая представляет важнейшую часть философии программирования C++ в части управления памятью. (См. врезку "Указатели и философия C++" ниже.)

Указатели и философия C++

Объектно-ориентированное программирование (ООП) отличается от традиционного процедурного программирования в том, что ООП делает особый акцент на принятии решений во время выполнения вместо времени компиляции. *Время выполнения* означает период работы программы, а *время компиляции* — период сборки программы компилятором в единое целое. Решения, принимаемые во время выполнения — это вроде того, как, будучи в отпуске, вы принимаете решение о том, какие достопримечательности стоит осмотреть, в зависимости от погоды и вашего настроения, в то время как решения, принимаемые во время компиляции, больше похожи на следование заранее разработанному плану, вне зависимости от любых условий.

Решения времени выполнения обеспечивают гибкость, позволяющую программе приспосабливаться к текущим условиям. Например, рассмотрим выделение памяти для массива. Традиционный способ предполагает объявление массива. Чтобы объявить массив в C++, вы должны заранее решить, какого он должен быть размера. Таким образом, размер массива устанавливается во время компиляции программы, т.е. это решение времени компиляции.

Возможно, вы думаете, что массив из 20 элементов будет достаточным в течение 80% времени, но однажды программе понадобится разместить 200 элементов. Чтобы обезопасить себя, вы используете массив размером в 200 элементов. Это приводит к тому, что ваша программа большую часть времени расходует память впустую. ООП пытается сделать программы более гибкими, откладывая принятие таких решений на стадию выполнения. Таким образом, после того, как программа запущена, она самостоятельно сможет решить, когда ей нужно размещать 20 элементов, а когда 200.

Короче говоря, с помощью ООП вы можете сделать выяснение размера массива решением времени выполнения. Чтобы обеспечить такой подход, язык должен предоставлять возможность создавать массивы — или что-то им подобное — непосредственно во время работы программы. Как вы вскоре увидите, метод, используемый C++, включает применение ключевого слова `new` для запроса необходимого объема памяти и применение указателей для нахождения выделенной по запросу памяти.

Принятие решений во время выполнения не является уникальной особенностью ООП. Но язык C++ делает написание соответствующего кода более прямолинейным, чем это позволяет C.

Новая стратегия хранения данных изменяет трактовку местоположения как именованной величины, а значения — как производной величины. Для этого предусмотрен специальный тип переменной — *указатель*, который может хранить адрес значения. Таким образом, имя указателя представляет местоположение. Применяя операцию `*`, называемую *косвенным значением* или *операцией разыменования*, можно получить значение, хранящееся в указанном месте. (Да, это тот же символ `*`, который применяется для обозначения арифметической операции умножения; C++ использует контекст для определения того, что подразумевается в каждом конкретном случае — умножение или разыменование.) Предположим, например, что `manly` — это указатель. В таком случае `manly` представляет адрес, а `*manly` — значение, находящееся по этому адресу. Комбинация `*manly` становится эквивалентом простой переменной типа `int`. Эти идеи демонстрируются в листинге 4.15. Также там показано, как объявляется указатель.

Листинг 4.15. `pointer.cpp`

```
// pointer.cpp -- наша первая переменная-указатель
#include <iostream>
int main()
{
   using namespace std;
   int updates = 6;                    // объявление переменной
   int * p_updates;                    // объявление указателя на int

   p_updates = &updates;               // присвоить адрес int указателю

   // Выразить значения двумя способами
   cout << "Values: updates = " << updates;
   cout << ", *p_updates = " << *p_updates << endl;

   // Выразить адреса двумя способами
   cout << "Addresses: &updates = " << &updates;
   cout << ", p_updates = " << p_updates << endl;

   // Изменить значение через указатель
   *p_updates = *p_updates + 1;
   cout << "Now updates = " << updates << endl;
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.15:

```
Values: updates = 6, *p_updates = 6
Addresses: &updates = 0x0065fd48, p_updates = 0x0065fd48
Now updates = 7
```

Как видите, переменная updates типа int и переменная-указатель p_updates — это две стороны одной монеты. Переменная updates в первую очередь представляет значение, а для получения его адреса используется операция &, в то время как p_updates представляет адрес, а для получения значения применяется операция * (рис. 4.8.) Поскольку p_updates указывает на updates, конструкции *p_updates и updates полностью эквивалентны. Вы можете использовать *p_updates точно так же, как используете переменную типа int. Как показано в примере из листинга 4.15, можно даже присваивать значения *p_updates. Это изменяет значение указываемой переменной — updates.

```
int jumbo = 23;
int * pe = &jumbo;
```

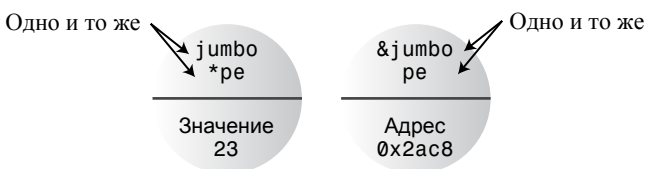


***Рис. 4.8.** Две стороны одной монеты*

Объявление и инициализация указателей

Давайте рассмотрим процесс объявления указателей. Компьютеру нужно отслеживать тип значения, на которое ссылается указатель. Например, адрес char обычно выглядит точно так же, как и адрес double, но char и double использует разное количество байт и разный внутренний формат представления значений. Поэтому объявление указателя должно задавать тип данных указываемого значения.

Например, предыдущий пример содержит следующее объявление:

```
int * p_updates;
```

Этот оператор устанавливает, что комбинация *p_updates имеет тип int.

Поскольку вы используете операцию *, применяя ее к указателю, сама переменная p_updates *должна быть* указателем. Мы говорим, что p_update указывает на тип int. Мы также говорим, что тип p_updates — это указатель на int, или точнее, int *. Итак, повторим еще раз: p_updates — это указатель (адрес), а *p_updates — это int, а не указатель (рис. 4.9). К слову, пробелы вокруг операции * не обязательны. Традиционно программисты на C используют следующую форму:

```
int *ptr;
```

Это подчеркивает идею, что комбинация *ptr является значением типа int. С другой стороны, многие программисты на C++ отдают предпочтение такой форме:

```
int* ptr;
```

Это подчеркивает идею о том, что int* — это тип "указатель на int". Для компилятора не важно, с какой стороны вы поместите пробел. Можно даже записать так:

```
int*ptr;
```

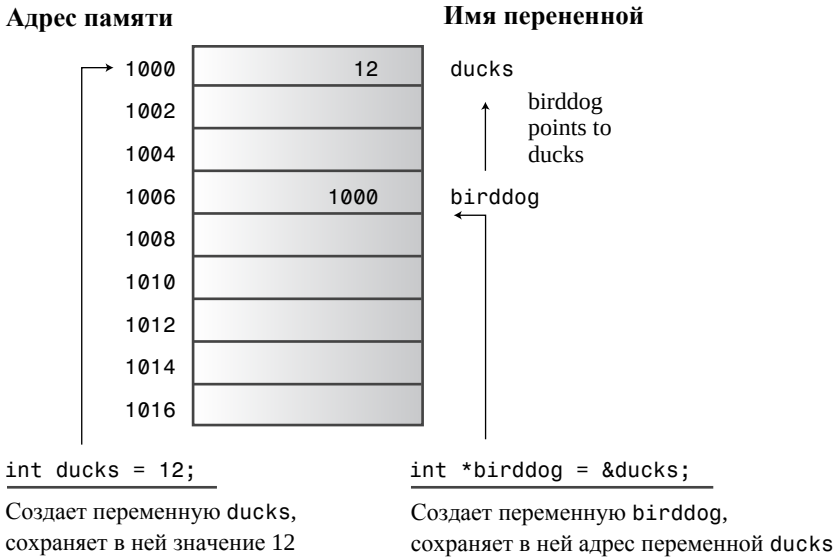


Рис. 4.9. Указатели хранят адреса

Однако учтите, что следующее объявление создает один указатель (`p1`) и одну обычную переменную типа `int` (`p2`):

```
int* p1, p2;
```

Знак `*` должен быть помещен возле каждой переменной типа указателя.

На заметку!

В языке C++ комбинация `int *` представляет составной тип "указатель на `int`".

Тот же самый синтаксис применяется для объявления указателей на другие типы:

```
double * tax_ptr;     // tax_ptr указывает на тип double
char * str;           // str указывает на тип char
```

Поскольку вы объявляете `tax_ptr` как указатель на `double`, компилятор знает, что `*tax_ptr` — это значение типа `double`. То есть ему известно, что `*tax_ptr` представляет число, сохраненное в формате с плавающей точкой и занимающее (в большинстве систем) 8 байт. Переменная указателя никогда не бывает просто указателем. Она всегда указывает на определенный тип. `tax_ptr` имеет тип "указатель на `double`" (или тип `double *`), а `str` — это тип "указатель на `char`" (или тип `char *`). Хотя оба они являются указателями, но указывают они на значения двух разных типов. Подобно массивам, указатели базируются на других типах.

Обратите внимание, что в то время как `tax_ptr` и `str` указывают на типы данных двух разных размеров, сами переменные `tax_ptr` и `str` обычно имеют одинаковый размер. То есть адрес `char` имеет тот же размер, что и адрес `double` — точно так же, как 1016 может быть номером дома, в котором располагается огромный склад, в то время как 1024 — номером небольшого коттеджа. Размер или значение адреса на самом деле ничего не говорят о том, какого вида и размера переменная или строение находится по этому адресу. Обычно адрес требует от 2 до 4 байт, в зависимости от компьютерной системы. (В некоторых системах могут применяться и более длинные адреса, а иногда система использует разный размер адресов для разных типов.)

Инициализировать указатель можно в операторе объявления. В этом случае инициализируется указатель, а не значение, на которое он указывает. То есть следующие операторы устанавливают pt, а не *pt равным значению &higgens:

```
int higgens = 5;
int * pt = &higgens;
```

В листинге 4.16 демонстрируется инициализация указателя определенным адресом.

Листинг 4.16. init_ptr.cpp

```
// init_ptr.cpp -- инициализация указателя
#include <iostream>
int main()
{
   using namespace std;
   int higgens = 5;
   int * pt = &higgens;

   cout << "Value of higgens = " << higgens
        << "; Address of higgens = " << &higgens << endl;
   cout << "Value of *pt = " << *pt
        << "; Value of pt = " << pt << endl;
   return 0;
}
```

Ниже показан вывод программы из листинга 4.16:

```
Value of higgens = 5; Address of higgens = 0012FED4
Value of *pt = 5; Value of pt = 0012FED4
```

Как видите, программа инициализирует pt, а не *pt, адресом переменной higgens. (Скорее всего, вы получите в своей системе другое значение адреса и, возможно, отображенное в другом формате.)

Опасность, связанная с указателями

Опасность подстерегает тех, кто использует указатели неосмотрительно. Очень важно понять, что при создании указателя в коде C++ компьютер выделяет память для хранения адреса, но не выделяет памяти для хранения данных, на которые указывает этот адрес. Выделение места для данных требует отдельного шага. Если пропустить этот шаг, как в следующем фрагменте, то это обеспечит прямой путь к проблемам:

```
long * fellow;           // создать указатель на long
*fellow = 223323;        // поместить значение в неизвестное место
```

Конечно, fellow — это указатель. Но на что он указывает? Никакого адреса переменной fellow в коде не присвоено. Так куда будет помещено значение 223323? Ответить на это невозможно. Поскольку переменная fellow не была инициализирована, она может иметь какое угодно значение. Что бы в ней ни содержалось, программа будет интерпретировать это как адрес, куда и поместит 223323. Если так случится, что fellow будет иметь значение 1200, компьютер попытается поместить данные по адресу 1200, даже если этот адрес окажется в середине вашего программного кода. На что бы ни указывал fellow, скорее всего, это будет не то место, куда вы хотели бы поместить число 223323. Ошибки подобного рода порождают самое непредсказуемое поведение программы и такие ошибки очень трудно отследить.

Внимание!
Золотое правило указателей: *всегда* инициализируйте указатель, чтобы определить точный и правильный адрес, прежде чем применять к нему операцию разыменования (*).

Указатели и числа

Указатели — это не целочисленные типы, даже несмотря на то, что компьютеры обычно выражают адреса целыми числами. Концептуально указатели представляют собой типы, отличные от целочисленных. Целочисленные значения можно суммировать, вычитать, умножать, делить и т.д. Но указатели описывают местоположение, и не имеет смысла, например, перемножать между собой два местоположения. В терминах допустимых над ними операций указатели и целочисленные типы отличаются друг от друга. Следовательно, нельзя просто присвоить целочисленное значение указателю:

```
int * pt;
pt = 0xB8000000;                // несоответствие типов
```

Здесь в левой части находится указатель на int, поэтому ему можно присваивать адрес, но в правой части задано просто целое число. Вы можете точно сказать, что 0xB8000000 — комбинация сегмент-смещение адреса видеопамяти в устаревшей системе, но этот оператор ничего не говорит программе о том, что данное число является адресом. В языке C до появления C99 подобного рода присваивания были разрешены. Однако в C++ применяются более строгие соглашения о типах, и компилятор выдаст сообщение об ошибке, говорящее о несоответствии типов. Если вы хотите использовать числовое значение в качестве адреса, то должны выполнить приведение типа, чтобы преобразовать числовое значение к соответствующему типу адреса:

```
int * pt;
pt = (int *) 0xB8000000;        // теперь типы соответствуют
```

Теперь обе стороны оператора присваивания представляют адреса, поэтому такое присваивание будет допустимым. Обратите внимание, что если есть значение адреса типа int, это не значит, что сам pt имеет тип int. Например, может существовать платформа, в которой тип int является двухбайтовым значением, то время как адрес — четырехбайтовым значением.

Указатели обладают и рядом других интересных свойств, которые мы обсудим, когда доберемся до соответствующей темы. А пока давайте посмотрим, как указатели могут использоваться для управления выделением памяти во время выполнения.

Выделение памяти с помощью операции new

Теперь, когда вы получили представление о работе указателей, давайте посмотрим, как с их помощью можно реализовать важнейший прием выделения памяти во время выполнения программы. До сих пор мы инициализировали указатели адресами переменных; переменные — это *именованная* память, выделенная во время компиляции, и каждый указатель, до сих пор использованный в примерах, просто представлял собой псевдоним для памяти, доступ к которой и так был возможен по именам переменных. Реальная ценность указателей проявляется тогда, когда во время выполнения выделяются *неименованные* области памяти для хранения значений. В этом случае указатели становятся единственным способом доступа к такой памяти. В языке C память можно выделять с помощью библиотечной функции malloc(). Ее можно применять и в C++, но язык C++ также предлагает лучший способ — операцию new.

Давайте испытаем этот новый прием, создав неименованное хранилище времени выполнения для значения типа int и обеспечив к нему доступ через указатель. Ключом ко всему является операция new. Вы сообщаете new, для какого типа данных запрашивается память; new находит блок памяти нужного размера и возвращает его адрес. Вы присваиваете этот адрес указателю, и на этом все. Ниже показан пример:

```
int * pn = new int;
```

Часть new int сообщает программе, что требуется некоторое новое хранилище, подходящее для хранения int. Операция new использует тип для того, чтобы определить, сколько байт необходимо выделить. Затем она находит память и возвращает адрес. Далее вы присваиваете адрес переменной pn, которая объявлена как указатель на int. Теперь pn — адрес, а *pn — значение, хранящееся по этому адресу. Сравните это с присваиванием адреса переменной указателю:

```
int higgens;
int * pt = &higgens;
```

В обоих случаях (pn и pt) вы присваиваете адрес значения int указателю. Во втором случае вы также можете обратиться к int по имени higgens. В первом случае доступ возможен только через указатель. Возникает вопрос: поскольку память, на которую указывает pn, не имеет имени, как обращаться к ней? Мы говорим, что pn указывает на *объект данных*. Это не "объект" в терминологии объектно-ориентированного программирования. Это просто объект, в смысле "вещь". Термин "объект данных" является более общим, чем "переменная", потому что он означает любой блок памяти, выделенный для элемента данных. Таким образом, переменная — это тоже объект, но память, на которую указывает pn, не является переменной. Метод обращения к объектам данных через указатель может показаться поначалу несколько запутанным, однако он обеспечивает программе высокую степень управления памятью.

Общая форма получения и назначения памяти отдельному объекту данных, который может быть как структурой, так и фундаментальным типом, выглядит следующим образом:

```
имяТипа * имя_указателя = new имяТипа;
```

Тип данных используется дважды: один раз для указания разновидности запрашиваемой памяти, а второй — для объявления подходящего указателя. Разумеется, если вы уже ранее объявили указатель на корректный тип, то можете его применить вместо объявления еще одного. В листинге 4.17 демонстрируется применение new для двух разных типов.

Листинг 4.17. use_new.cpp

```
// use_new.cpp -- использование операции new
#include <iostream>
int main()
{
   using namespace std;
   int nights = 1001;
   int * pt = new int;         // выделение пространства для int
   *pt = 1001;                 // сохранение в нем значения

   cout << "nights value = ";                          // значение nights
   cout << nights << ": location " << &nights << endl; // расположение nights
   cout << "int ";                                     // значение и расположение int
   cout << "value = " << *pt << ": location = " << pt << endl;
   double * pd = new double;   // выделение пространства для double
   *pd = 10000001.0;           // сохранение в нем значения double
```

```
    cout << "double ";
    cout << "value = " << *pd << ": location = " << pd << endl;
          // значение и расположение double
    cout << "location of pointer pd: " << &pd << endl;
          // расположение указателя pd
    cout << "size of pt = " << sizeof(pt);                    // размер pt
    cout << ": size of *pt = " << sizeof(*pt) << endl;       // размер *pt
    cout << "size of pd = " << sizeof pd;                     // размер pd
    cout << ": size of *pd = " << sizeof(*pd) << endl;       // размер *pd
    return 0;
}
```

Ниже показан вывод программы из листинга 4.17:

```
nights value = 1001: location 0028F7F8
int value = 1001: location = 00033A98
double value = 1e+007: location = 000339B8
location of pointer pd: 0028F7FC
size of pt = 4: size of *pt = 4
size of pd = 4: size of *pd = 8
```

Естественно, точные значения адресов памяти будут варьироваться от системы к системе.

Замечания по программе

Программа в листинге 4.17 использует операцию `new` для выделения памяти под объекты данных типа `int` и типа `double`. Это происходит во время выполнения программы. Указатели `pt` и `pd` указывают на эти объекты данных. Без них вы не смогли бы получить к ним доступ. С ними же вы можете применять `*pt` и `*pd` подобно тому, как вы используете обычные переменные. Вы присваиваете значения новым объектам данных, присваивая их `*pt` и `*pd`. Аналогично вы посылаете на печать `*pt` и `*pd`, чтобы отобразить эти значения.

Программа в листинге 4.17 также демонстрирует одну из причин того, что необходимо объявить тип данных, на которые указывает указатель. Сам по себе адрес относится только к началу сохраненного объекта, он не включает информации о типе или размере. К тому же обратите внимание, что указатель на `int` имеет тот же размер, что и указатель на `double`. Оба они являются адресами. Но поскольку в `use_new.cpp` объявлены типы указателей, программа знает, что `*pd` имеет тип `double` размером в 8 байт, в то время как `*pt` представляет собой значение типа `int` размером в 4 байта. Когда `use_new.cpp` печатает значение `*pd`, `cout` известно, сколько байт нужно прочитать и как их интерпретировать.

Еще один момент, который следует отметить, состоит в том, что обычно операция `new` использует другие блоки памяти, чем применяемые ранее объявления обычных переменных. Переменные `nights` и `pd` хранят свои значения в области памяти под названием *стек*, тогда как память, выделяемая операцией `new`, находится в области, называемой *кучей* или *свободным хранилищем*. Дополнительные сведения об этом можно почерпнуть в главе 9.

Нехватка памяти?

Может случиться так, что у компьютера не окажется достаточно доступной памяти, чтобы удовлетворить запрос `new`. Когда такое происходит, операция `new` обычно реагирует генерацией исключения; способы обработки ошибок рассматриваются в главе 15. В более старых реализациях `new` возвращает значение `0`.

В C++ указатель со значением 0 называется *null-указателем* (нулевым указателем). C++ гарантирует, что нулевой указатель никогда не указывает на допустимые данные, поэтому он часто используется в качестве признака неудачного завершения операций или функций, которые в противном случае должны возвращать корректные указатели. Оператор if, описанный в главе 6, поможет справиться с подобной ситуацией. А сейчас просто важно знать, что в C++ предлагаются инструменты для обнаружения и реагирования на сбои при выделении памяти.

Освобождение памяти с помощью операции delete

Использование операции new для запрашивания памяти, когда она нужна — одна из сторон пакета управления памятью C++. Второй стороной является операция delete, которая позволяет вернуть память в пул свободной памяти, когда работа с ней завершена. Это — важный шаг к максимально эффективному использованию памяти. Память, которую вы возвращаете, или *освобождаете*, затем может быть повторно использована другими частями программы. Операция delete применяется с указателем на блок памяти, который был выделен операцией new:

```
int * ps = new int;   // выделить память с помощью операции new
...                   // использовать память
delete ps;            // по завершении освободить память
                      // с помощью операции delete
```

Это освобождает память, на которую указывает ps, но не удаляет сам указатель ps. Вы можете повторно использовать ps — например, чтобы указать на другой выделенный new блок памяти. Вы всегда должны обеспечивать сбалансированное применение new и delete; в противном случае вы рискуете столкнуться с таким явлением, как *утечка памяти*, т.е. ситуацией, когда память выделена, но более не может быть использована. Если утечки памяти слишком велики, то попытка программы выделить очередной блок может привести к ее аварийному завершению.

Вы не должны пытаться освобождать блок памяти, который уже был однажды освобожден. Стандарт C++ гласит, что результат таких попыток не определен, а это значит, что последствия могут оказаться любыми. Кроме того, вы не можете с помощью операции delete освобождать память, которая была выделена посредством объявления обычных переменных:

```
int * ps = new int;   // нормально
delete ps;            // нормально
delete ps;            // теперь не нормально!
int jugs = 5;         // нормально
int * pi = &jugs;     // нормально
delete pi;            // не допускается, память не была выделена new
```

Внимание!

Операция delete должна использоваться только для освобождения памяти, выделенной с помощью new. Однако применение delete к нулевому указателю вполне безопасно.

Обратите внимание, что обязательным условием применения операции delete является использование ее с памятью, выделенной операцией new. Это не значит, что вы обязаны применять тот же указатель, который был использован с new - просто нужно задать тот же адрес:

```
int * ps = new int;   // выделение памяти
int * pq = ps;        // установка второго указателя на тот же блок
delete pq;            // вызов delete для второго указателя
```

Обычно не стоит создавать два указателя на один и тот же блок памяти, т.к. это может привести к ошибочной попытке освобождения одного и того же блока дважды. Но, как вы вскоре убедитесь, применение второго указателя оправдано, когда вы работаете с функциями, возвращающими указатель.

Использование операции new для создания динамических массивов

Если все, что нужно программе — это единственное значение, вы можете объявить обычную переменную, поскольку это намного проще (хотя и не так впечатляет), чем применение new для управления единственным небольшим объектом данных. Использование операции new более типично с крупными фрагментами данных, такими как массивы, строки и структуры. Именно в таких случаях операция new является полезной. Предположим, например, что вы пишете программу, которой может понадобиться массив, а может, и нет — это зависит от информации, поступающий во время выполнения. Если вы создаете массив простым объявлением, пространство для него распределяется раз и навсегда — во время компиляции. Будет ли востребованным массив в программе или нет — он все равно существует и занимает место в памяти. Распределение массива во время компиляции называется *статическим связыванием* и означает, что массив встраивается в программу во время компиляции. Но с помощью new вы можете создать массив, когда это необходимо, во время выполнения программы, либо не создавать его, если потребность в нем отсутствует. Или же вы можете выбрать размер массива уже после того, как программа запущена. Это называется *динамическим связыванием* и означает, что массив будет создан во время выполнения программы. Такой массив называется *динамическим массивом*. При статическом связывании вы должны жестко закодировать размер массива во время написания программы. При динамическом связывании программа может принять решение о размере массива во время своей работы.

Пока что мы рассмотрим два важных обстоятельства относительно динамических массивов: как применять операцию new для создания массива и как использовать указатель для доступа к его элементам.

Создание динамического массива с помощью операции new

Создать динамический массив на C++ легко; вы сообщаете операции new тип элементов массива и требуемое количество элементов. Синтаксис, необходимый для этого, предусматривает указание имени типа с количеством элементов в квадратных скобках. Например, если необходим массив из 10 элементов int, следует записать так:

```
int * psome = new int [10]; // получение блока памяти из 10 элементов типа int
```

Операция new возвращает адрес первого элемента в блоке. В данном примере это значение присваивается указателю psome.

Как всегда, вы должны сбалансировать каждый вызов new соответствующим вызовом delete, когда программа завершает работу с этим блоком памяти. Однако использование new с квадратными скобками для создания массива требует применения альтернативной формы delete при освобождении массива:

```
delete [] psome;          // освобождение динамического массива
```

Присутствие квадратных скобок сообщает программе, что она должна освободить весь массив, а не только один элемент, на который указывает указатель. Обратите внимание, что скобки расположены между delete и указателем.

Если вы используете new без скобок, то и соответствующая операция delete тоже должна быть без скобок. Если же new со скобками, то и соответствующая операция delete должна быть со скобками. Ранние версии C++ могут не распознавать нотацию с квадратными скобками. Согласно стандарту ANSI/ISO, однако, эффект от несоответствия форма new и delete не определен, т.е. вы не должны рассчитывать в этом случае на какое-то определенное поведение. Вот пример:

```
int * pt = new int;
short * ps = new short [500];
delete [] pt;                  // эффект не определен, не делайте так
delete ps;                     // эффект не определен, не делайте так
```

Короче говоря, при использовании new и delete необходимо придерживаться перечисленных ниже правил.

- Не использовать delete для освобождения той памяти, которая не была выделена new.
- Не использовать delete для освобождения одного и того же блока памяти дважды.
- Использовать delete[], если применялась операция new[] для размещения массива.
- Использовать delete без скобок, если применялась операция new для размещения отдельного элемента.
- Помнить о том, что применение delete к нулевому указателю является безопасным (при этом ничего не происходит).

Теперь вернемся к динамическому массиву. Обратите внимание, что psome — это указатель на отдельное значение int, являющееся первым элементом блока. Отслеживать количество элементов в блоке возлагается на вас как разработчика. То есть, поскольку компилятор не знает о том, что psome указывает на первое из 10 целочисленных значений, вы должны писать свою программу так, чтобы она самостоятельно отслеживала количество элементов.

На самом деле программе, конечно же, известен объем выделенной памяти, так что она может корректно освободить ее позднее, когда вы воспользуетесь операцией delete []. Однако эта информация не является открытой; вы, например, не можете применить операцию sizeof, чтобы узнать количество байт в выделенном блоке.

Общая форма выделения и назначения памяти для массива выглядит следующим образом:

```
имя_типа имя_указателя = new имя_типа [количество_элементов];
```

Вызов операции new выделяет достаточно большой блок памяти, чтобы в нем поместилось *количество_элементов* элементов типа *имя_типа*, и устанавливает в *имя_указателя* указатель на первый элемент. Как вы вскоре увидите, *имя_указателя* можно использовать точно так же, как обычное имя массива.

Использование динамического массива

Как работать с динамическим массивом после его создания? Для начала подумаем о проблеме концептуально. Следующий оператор создает указатель psome, который указывает на первый элемент блока из 10 значений int:

```
int * psome = new int [10];   // получить блок для 10 элементов типа int
```

Представьте его как палец, указывающий на первый элемент. Предположим, что int занимает 4 байта. Перемещая палец на 4 байта в правильном направлении, вы можете указать на второй элемент. Всего имеется 10 элементов, что является допустимым диапазоном, в пределах которого можно передвигать палец. Таким образом, операция new снабжает всей необходимой информацией для идентификации каждого элемента в блоке.

Теперь взглянем на проблему практически. Как можно получить доступ к этим элементам? С первым элементом проблем нет. Поскольку psome указывает на первый элемент массива, то *psome и есть значение первого элемента. Но остается еще девять элементов. Простейший способ доступа к этим элементам может стать сюрпризом для вас, если вы не работали с языком C; просто используйте указатель, как если бы он был именем массива. То есть можно писать psome[0] вместо *psome для первого элемента, psome[1] — для второго и т.д. Получается, что применять указатель для доступа к динамическому массиву очень просто, хотя не вполне понятно, почему этот метод работает. Причина в том, что C и C++ внутренне все равно работают с массивами через указатели. Подобная эквивалентность указателей и массивов — одно из замечательных свойств C и C++. (Иногда это также и проблема, но это уже другая история.) Ниже об этом речь пойдет более подробно. В листинге 4.18 показано, как использовать new для создания динамического массива и доступа к его элементам с применением нотации обычного массива. В нем также отмечается фундаментальное отличие между указателем и реальным именем массива.

Листинг 4.18. arraynew.cpp

```
// arraynew.cpp -- использование операции new для массивов
#include <iostream>
int main()
{
    using namespace std;
    double * p3 = new double [3];   // пространство для 3 значений double
    p3[0] = 0.2;                    // трактовать p3 как имя массива
    p3[1] = 0.5;
    p3[2] = 0.8;

    cout << "p3[1] is " << p3[1] << ".\n";          // вывод p3[1]
    p3 = p3 + 1;                                    // увеличение указателя
    cout << "Now p3[0] is " << p3[0] << " and ";    // вывод p3[0]
    cout << "p3[1] is " << p3[1] << ".\n";          // вывод p3[1]
    p3 = p3 - 1;                                    // возврат указателя в начало
    delete [] p3;                                   // освобождение памяти
    return 0;
}
```

Ниже показан вывод программы из листинга 4.18:

```
p3[1] is 0.5.
Now p3[0] is 0.5 and p3[1] is 0.8.
```

Как видите, arraynew.cpp использует указатель p3, как если бы он был именем массива: p3[0] для первого элемента и т.д. Фундаментальное отличие между именем массива и указателем проявляется в следующей строке:

```
p3 = p3 + 1;  // допускается для указателей, но не для имен массивов
```

Вы не можете изменить значение для имени массива. Но указатель — переменная, а потому ее значение можно изменить. Обратите внимание на эффект от добавле-

ния 1 к p3. Теперь выражение p3[0] ссылается на бывший второй элемент массива. То есть добавление 1 к p3 заставляет p3 указывать на второй элемент вместо первого. Вычитание 1 из значения указателя возвращает его назад, в исходное значение, поэтому программа может применить delete[] с корректным адресом.

Действительные адреса соседних элементов int отличаются на 2 или 4 байта, поэтому тот факт, что добавление 1 к p3 дает адрес следующего элемента, говорит о том, что арифметика указателей устроена специальным образом. Так оно и есть на самом деле.

Указатели, массивы и арифметика указателей

Родство указателей и имен массивов происходит из *арифметики указателей*, а также того, как язык C++ внутренне работает с массивами. Сначала рассмотрим арифметику указателей. Добавление единицы к целочисленной переменной увеличивает ее значение на единицу, но добавление единицы к переменной типа указателя увеличивает ее значение на количество байт, составляющих размер типа, на который она указывает. Добавление единицы к указателю на double добавляет 8 байт к числовой величине указателя на системах с 8-байтным double, в то время как добавление единицы к указателю на short добавляет к его значению 2 байта. Код в листинге 4.19 доказывает истинность этого утверждения. Он также демонстрирует еще один важный момент: C++ интерпретирует имена массивов как адреса.

Листинг 4.19. addpntrs.cpp

```
// addpntrs.cpp -- сложение указателей
#include <iostream>
int main()
{
    using namespace std;
    double wages[3] = {10000.0, 20000.0, 30000.0};
    short stacks[3] = {3, 2, 1};

    // Два способа получить адрес массива
    double * pw = wages;      // имя массива равно адресу
    short * ps = &stacks[0]; // либо использование операции взятия адреса
                              // с элементом массива
    cout << "pw = " << pw << ", *pw = " << *pw << endl;
    pw = pw + 1;
    cout << "add 1 to the pw pointer:\n";             // добавление 1 к указателю pw
    cout << "pw = " << pw << ", *pw = " << *pw << "\n\n";
    cout << "ps = " << ps << ", *ps = " << *ps << endl;
    ps = ps + 1;
    cout << "add 1 to the ps pointer:\n";             // добавление 1 к указателю ps
    cout << "ps = " << ps << ", *ps = " << *ps << "\n\n";

    // Доступ к двум элементам с помощью нотации массивов
    cout << "access two elements with array notation\n";
    cout << "stacks[0] = " << stacks[0]
         << ", stacks[1] = " << stacks[1] << endl;

    // Доступ к двум элементам с помощью нотации указателей
    cout << "access two elements with pointer notation\n";
    cout << "*stacks = " << *stacks
         << ", *(stacks + 1) = " << *(stacks + 1) << endl;
    cout << sizeof(wages) << " = size of wages array\n"; // размер массива wages
    cout << sizeof(pw) << " = size of pw pointer\n";     // размер указателя pw
    return 0;
}
```

Ниже приведен вывод программы из листинга 4.19:

```
pw = 0x28ccf0, *pw = 10000
add 1 to the pw pointer:
pw = 0x28ccf8, *pw = 20000

ps = 0x28ccea, *ps = 3
add 1 to the ps pointer:
ps = 0x28ccec, *ps = 2

access two elements with array notation
stacks[0] = 3, stacks[1] = 2
access two elements with pointer notation
*stacks = 3, *(stacks + 1) = 2
24 = size of wages array
4 = size of pw pointer
```

Замечания по программе

В большинстве контекстов C++ интерпретирует имя массива как адрес его первого элемента. Таким образом, следующий оператор создает `pw` как указатель на тип `double`, затем инициализирует его `wages`, который также является адресом первого элемента массива `wages`:

```
double * pw = wages;
```

Для `wages`, как и любого другого массива, справедливо следующее утверждение:

```
wages = &wages[0] = адрес первого элемента массива
```

Чтобы доказать, что это так, программа явно использует операцию взятия адреса в выражении `&stacks[0]` для инициализации указателя `ps` адресом первого элемента массива `stacks`.

Далее программа инспектирует значения `pw` и `*pw`. Первое из них представляет адрес, а второе — значение, расположенное по этому адресу. Поскольку `pw` указывает на первый элемент, значение, отображаемое `*pw`, и будет значением первого элемента — `10000`. Затем программа прибавляет единицу к `pw`. Как и ожидалось, это добавляет 8 (`fd24 + 8 = fd2c` в шестнадцатеричном виде) к числовому значению адреса, потому что `double` в этой системе занимает 8 байт. Это присваивает `pw` адрес второго элемента. Таким образом, теперь `*pw` равно `20000`, т.е. значение второго элемента (рис. 4.10). (Значения адресов на рисунке подкорректированы для ясности.)

После этого программа выполняет аналогичные шаги для `ps`. На этот раз, поскольку `ps` указывает на тип `short`, а размер значений `short` составляет 2 байтам, добавление 1 к этому указателю увеличивает его значение на 2 (`0x28ccea + 2 = 0x28ccec` в шестнадцатеричной форме). Опять-таки, в результате указатель устанавливается на следующий элемент массива.

На заметку!
Добавление единицы к переменной указателя увеличивает его значение на количество байт, представляющее размер типа, на который он указывает.

Теперь рассмотрим выражение `stacks[1]`. Компилятор C++ трактует это выражение точно так же, как если бы вы написали `*(stacks + 1)`. Второе выражение означает вычисление адреса второго элемента массива, и затем извлечение значения, сохраненного в нем. Конечный результат — значение `stacks[1]`. (Приоритет операций требует применения скобок. Без них значение `1` было бы добавлено к `*stacks` вместо `stacks`.)

```
double wages[3] = {10000.0, 20000.0, 30000.0};
short stacks[3] = {3, 2, 1};
double * pw = wages;
short * ps = &stacks[0];
```

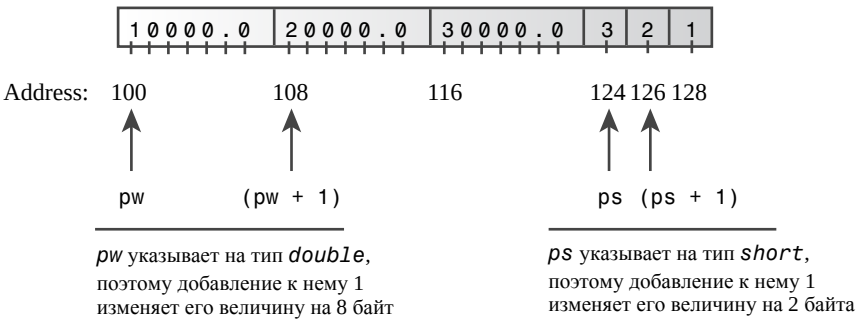


***Рис. 4.10.** Сложение указателей*

Вывод программы демонстрирует, что `*(stacks + 1)` и `stacks[1]` — это одно и то же. Аналогично `*(stacks + 1)` эквивалентно `stacks[2]`. В общем случае, всякий раз, когда вы используете нотацию массивов, C++ выполняет следующее преобразование:

имя_массива`[i]` превращается в `*(`*имя_массива*` + i)`

И если вы используете указатель вместо имени массива, C++ осуществляет то же самое преобразование:

имя_указателя`[i]` превращается в `*(`*имя_указателя*` + i)`

Таким образом, во многих отношениях имена указателей и имена массивов можно использовать одинаковым образом. Нотация квадратных скобок применима и там, и там. К обоим можно применять операцию разыменования (`*`). В большинстве выражений каждое имя представляет адрес. Единственное отличие состоит в том, что значение указателя изменить можно, а имя массива является константой:

```
имя_указателя = имя_указателя + 1;      // правильно
имя_массива = имя_массива + 1;          // не допускается
```

Второе отличие заключается в том, что применение операции `sizeof` к имени массива возвращает размер массива в байтах, но применение `sizeof` к указателю возвращает размер указателя, даже если он указывает на массив. Например, в листинге 4.19 как `pw`, так и `wages` ссылаются на один и тот же массив. Однако применение операции `sizeof` к ним порождает разные результаты:

24 = размер массива `wages` ← отображение `sizeof` для `wages`
4 = размер указателя `pw` ← отображение `sizeof` для `pw`

Это один из случаев, когда C++ не интерпретирует имя массива как адрес.

Адрес массива

Получение адреса массива является другим случаем, при котором имя массива не интерпретируется как его адрес. Но подождите, разве имя массива не интерпретируется как адрес массива? Не совсем — имя массива интерпретируется как адрес первого элемента массива, в то время как применение операции взятия адреса приводит к выдаче адреса целого массива:

```
short tell[10];           // создание массива из 20 байт
cout << tell << endl;     // отображение &tell[0]
cout << &tell << endl;    // отображение адреса целого массива
```

С точки зрения числового представления эти два адреса одинаковы, но концептуально `&tell[0]` и, следовательно, `tell` — это адрес 2-байтного блока памяти, тогда как `&tell` — адрес 20-байтного блока памяти. Таким образом, выражение `tell + 1` приводит к добавлению `2` к значению адреса, а `&tell + 1` — к добавлению `20` к значению адреса. Это можно выразить и по-другому: `tell` имеет тип "указатель на `short`", или `short *`, а `&tell` — тип "указатель на массив из 20 элементов `short`", или `short (*)[20]`.

Теперь вас может заинтересовать происхождение последнего описания типа. Сначала посмотрим, как можно объявить и инициализировать указатель этого типа:

```
short (*pas)[20] = &tell;  // pas указывает на массив из 20 элементов short
```

Если опустить круглые скобки, то правила приоритетов будут ассоциировать `[20]` в первую очередь с `pas`, делая `pas` массивом из 20 указателей на `short`, поэтому круглые скобки необходимы. Далее, если вы хотите описать тип переменной, вы можете воспользоваться объявлением этой переменной в качестве руководства и удалить имя переменной. Таким образом, типом `pas` является `short (*)[20]`. Кроме того, обратите внимание, что поскольку значение `pas` установлено в `&tell`, `*pas` эквивалентно `tell`, и `(*pas)[0]` будет первым элементом массива `tell`.

Говоря кратко, использовать `new` для создания массива и применять указатели для доступа к его различным элементам очень просто. Вы просто трактуете указатель как имя массива. Однако понять, почему это работает — интересная задача. Если вы действительно хотите понимать массивы и указатели, то должны тщательно исследовать их поведение, связанное с изменчивостью.

Подведение итогов относительно указателей

Позже вы сможете несколько углубить свои знания об указателях, а пока подведем итоги относительно того, что известно об указателях и массивах к настоящему моменту.

Объявление указателей

Чтобы объявить указатель на определенный тип, нужно использовать следующую форму:

```
имяТипа * имяУказателя;
```

Вот некоторые примеры:

```
double * pn;              // pn может указывать на значение double
char * pc;                // pc может указывать на значение char
```

Здесь `pn` и `pc` — указатели, а `double *` и `char *` — нотация C++ для представления указателя на `double` и указателя на `char`.

Присваивание значений указателям

Указателям должны быть присвоены адреса памяти. Можно применить операцию `&` к имени переменной, чтобы получить адрес именованной области памяти, либо операцию `new`, которая возвращает адрес неименованной памяти.

Вот некоторые примеры:

```
double * pn;              // pn может указывать на значение double
double * pa;              // так же и pa
char * pc;                // pc может указывать на значение char
```

```
double bubble = 3.2;
pn = &bubble;          // присваивание адреса bubble переменной pn
pc = new char;         // присваивание адреса выделенной памяти char
                       // переменной pc
pa = new double[30]; // присваивание адреса массива из 30 double переменной pa
```

Разыменование указателей

Разыменование указателя — это получение значения, на которое он указывает. Для этого к указателю применяется операция разыменования (*). То есть, если pn — указатель на bubble, как в предыдущем примере, то *pn — значение, на которое он указывает, в данном случае — 3.2.

Вот некоторые примеры:

```
cout << *pn;           // вывод значения bubble
*pc = 'S';             // помещение 'S' в область памяти, на которую указывает pc
```

Нотация массивов — второй способ разыменования указателя; например, pn[0] — это то же самое, что и *pn. Никогда не следует разыменовывать указатель, который не был инициализирован правильным адресом.

Различие между указателем и указываемым значением

Помните, если pt — указатель на int, то *pt — не указатель на int, а полный эквивалент переменной типа int. Указателем является просто pt.

Вот некоторые примеры:

```
int * pt = new int;    // присваивание адреса переменной pt
*pt = 5;               // сохранение 5 по этому адресу
```

Имена массивов

В большинстве контекстов C++ трактует имя массива как эквивалент адреса его первого элемента.

Вот пример:

```
int tacos[10];         // теперь tacos — то же самое, что и &tacos[0]
```

Одно исключение из этого правила связано с применением операции sizeof к имени массива. В этом случае sizeof возвращает размер всего массива в байтах.

Арифметика указателей

C++ позволяет добавлять целые числа к указателю. Результат добавления к указателю единицы равен исходному адресу плюс значение, эквивалентное количеству байт в указываемом объекте. Можно также вычесть один указатель из другого, чтобы получить разницу между двумя указателями. Последняя операция, которая возвращает целочисленное значение, имеет смысл только в случае, когда два указателя указывают на элементы одного и того же массива (указание одной из позиций за границей массива также допускается); при этом результат означает расстояние между элементами массива.

Ниже приведен ряд примеров:

```
int tacos[10] = {5,2,8,4,1,2,2,4,6,8};
int * pt = tacos;      // предположим, что pf и tacos указывают на адрес 3000
pt = pt + 1;           // теперь pt равно 3004, если int имеет размер 4 байта
int *pe = &tacos[9]; // pe равно 3036, если int имеет размер 4 байта
pe = pe - 1;           // теперь pe равно 3032 - адресу элемента tacos[8]
int diff = pe - pt;    // diff равно 7, т.е. расстоянию между tacos[8] и tacos[1]
```

Динамическое и статическое связывание для массивов

Объявление массива можно использовать для создания массива со статическим связыванием — т.е. массива, размер которого фиксирован на этапе компиляции:

```
int tacos[10]; // статическое связывание, размер фиксирован во время компиляции
```

Для создания массива с динамическим связыванием (динамического массива) используется операция `new[]`. Память для этого массива выделяется в соответствии с размером, указанным во время выполнения программы. Когда работа с таким массивом завершена, выделенная ему память освобождается с помощью операции `delete[]`:

```
int size;
cin >> size;
int * pz = new int [size];  // динамическое связывание, размер
                            // устанавливается во время выполнения
...
delete [] pz; // освобождение памяти по окончании работы с массивом
```

Нотация массивов и нотация указателей

Использование нотации массивов с квадратными скобками эквивалентно разыменованию указателя:

`tacos[0]` означает `*tacos` и означает значение, находящееся по адресу `tacos`

`tacos[3]` означает `*(tacos+3)` и означает значение, находящееся по адресу `tacos+3`

Это справедливо как для имен массивов, так и для переменных типа указателей, поэтому вы можете применять обе нотации.

Вот некоторые примеры:

```
int * pt = new int [10];  // pt указывает на блок из 10 значений int
*pt = 5;                  // присваивает элементу 0 значение 5
pt[0] = 6;                // присваивает элементу 0 значение 6
pt[9] = 44;               // устанавливает десятый элемент (элемент номер 9) в 44
int coats[10];
*(coats + 4) = 12;        // устанавливает coats[4] в 12
```

Указатели и строки

Специальные отношения между массивами и указателями расширяют строки в стиле C. Рассмотрим следующий код:

```
char flower[10] = "rose";
cout << flower << "s are red\n";
```

Имя массива — это адрес его первого элемента, потому `flower` в операторе `cout` представляет адрес элемента `char`, содержащего символ `r`. Объект `cout` предполагает, что адрес `char` — это адрес строки, поэтому печатает символ, расположенный по этому адресу, и затем продолжает печать последующих символов до тех пор, пока не встретит нулевой символ (`\0`). Короче говоря, вы сообщаете `cout` адрес символа, он печатает все, что находится в памяти, начиная с этого символа и до нулевого.

Ключевым фактором здесь является не то, что `flower` — имя массива, а то, что `flower` трактуется как адрес значения `char`. Это предполагает, что вы можете использовать переменную-указатель на `char` в качестве аргумента для `cout`, потому что она тоже содержит адрес `char`. Разумеется, этот указатель должен указывать на начало строки. Чуть позже мы проверим это.

Но как насчет финальной части предыдущего оператора cout? Если flower — на самом деле адрес первого символа строки, что собой представляет выражение "s are red\n"? Согласно принципам, которыми руководствуется cout при выводе строк, эта строка в кавычках также должна быть адресом. Так оно и есть: в C++ строка в кавычках, как и имя массива, служит адресом его первого элемента. Предыдущий код на самом деле не посылает полную строку объекту cout; он просто передает адрес строки. Это значит, что строки в массиве, строковые константы в кавычках и строки, описываемые указателями — все обрабатываются одинаково. Каждая из них передается в виде адреса. Конечно, это уменьшает объем работ компьютеру по сравнению с тем, который потребовался бы в случае передачи каждого символа строки.

На заметку!

С объектом cout, как и в большинстве других выражений C++, имя массива char, указатель на char, а также строковая константа в кавычках — все интерпретируются как адрес первого символа строки.

В листинге 4.20 иллюстрируется применение различных форм строк. Код в этом листинге использует две функции из библиотеки обработки строк. Функция strlen(), которую вы уже применяли ранее, возвращает длину строки. Функция strcpy() копирует строку из одного места в другое. Обе функции имеют прототипы в файле заголовков cstring (или string.h — в устаревших реализациях). В программе также присутствуют комментарии, предупреждающие о возможных случаях неправильного применения указателей, которых следует избегать.

Листинг 4.20. ptrstr.cpp

```
// ptrstr.cpp -- использование указателей на строки
#include <iostream>
#include <cstring>  // объявление strlen(), strcpy()
int main()
{
   using namespace std;
   char animal[20] = "bear";           // animal содержит bear
   const char * bird = "wren";         // bird содержит адрес строки
   char * ps;                          // не инициализировано
   cout << animal << " and ";          // отображение bear
   cout << bird << "\n";               // отображение wren
   // cout << ps << "\n";              // может отобразить мусор, но может вызвать
                                       // и аварийное завершение программы
   cout << "Enter a kind of animal: ";
   cin >> animal;                      // нормально, если вводится меньше 20 символов
   // cin >> ps; очень опасная ошибка, чтобы пробовать;
   // ps не указывает на выделенное пространство

   ps = animal;                        // установка ps в указатель на строку
   cout << ps << "!\n";                // нормально; то же, что и применение animal
   cout << "Before using strcpy():\n";
   cout << animal << " at " << (int *) animal << endl;
   cout << ps << " at " << (int *) ps << endl;
   ps = new char[strlen(animal) + 1];    // получение нового хранилища
   strcpy(ps, animal);                   // копирование строки в новое хранилище
   cout << "After using strcpy():\n";
   cout << animal << " at " << (int *) animal << endl;
   cout << ps << " at " << (int *) ps << endl;
   delete [] ps;
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.20:

```
bear and wren
Enter a kind of animal: fox
fox!
Before using strcpy():
fox at 0x0065fd30
fox at 0x0065fd30
After using strcpy():
fox at 0x0065fd30
fox at 0x004301c8
```

Замечания по программе

Программа в листинге 4.20 создает один массив `char` (`animal`) и две переменных типа "указатель на `char`" (`bird` и `ps`). Программа начинается с инициализации массива `animal` строкой `"bear"` — точно так же, как вы инициализировали массивы и раньше. Затем программа делает нечто новое. Она инициализирует указатель на `char` строкой:

```
const char * bird = "wren";  // bird содержит адрес строки
```

Вспомните, что `"wren"` на самом деле представляет собой адрес строки, поэтому приведенный выше оператор присваивает адрес `"wren"` указателю `bird`. (Обычно компилятор выделяет область памяти для размещения строк в кавычках, указанных в исходном коде, ассоциируя каждую сохраненную строку с ее адресом.) Это значит, что указатель `bird` можно применять так же, как использовалась бы строка `"wren"`, например:

```
cout << "A concerned " << bird << " speaks\n";
```

Строковые литералы являются константами; именно поэтому в объявлении присутствует ключевое слово `const`. Применение `const`, таким образом, означает, что вы можете использовать `bird` для доступа к строке, но не можете изменять ее. В главе 7 тема константных указателей рассматривается более подробно. И, наконец, указатель `ps` остается неинициализированным, поэтому он не может указывать ни на какую строку (Как вы знаете, это плохая идея, и данный пример — не исключение.)

Далее программа иллюстрирует тот факт, что имя массива `animal` и указатель `bird` можно использовать с `cout` совершенно одинаково. В конце концов, оба они являются адресами строк, и `cout` отображает две строки (`"bear"` и `"wren"`), расположенные по указанным адресам. Если вы удалите знаки комментария с кода, который пытается отобразить `ps`, то можете получить пустую строку, какой-нибудь мусор или даже привести программу к аварийному завершению. Создание неинициализированных указателей подобно выдаче незаполненного чека с подписью: вы утрачиваете контроль над его использованием.

Что касается ввода, то здесь ситуация несколько отличается. Использовать массив `animal` для ввода безопасно до тех пор, пока ввод достаточно краток, чтобы уместиться в отведенный массив. Однако было бы неправильно применять для ввода указатель `bird`.

- Некоторые компиляторы трактуют строковые литералы как константы, доступные только для чтения, что ведет к ошибкам времени выполнения при попытках записи в них данных. То, что строковые литералы являются константами — обязательное поведение C++, однако пока не все разработчики компиляторов отказались от старого подхода при их обработке.

- Некоторые компиляторы используют только одну копию строкового литерала для представления всех его вхождений в тексте программы.

Давайте проясним второй пункт. C++ не гарантирует уникальное сохранение строкового литерала. То есть, если вы используете литерал `"wren"` несколько раз в программе, компилятор может сохранить либо несколько копий этой строки, либо всего одну. Если он сохраняет одну, то установка в `bird` адреса `"wren"` заставляет его указать на единственную копию этой строки. Чтение нового значения в эту одну строку может повлиять на строки, которые изначально имели то же самое значение, но логически были совершенно независимыми, встречаясь в других местах программы. В любом случае, поскольку указатель `bird` объявлен как `const`, компилятор предотвратит любые попытки изменить содержимое, расположенное по адресу, на который указывает `bird`.

Еще хуже обстоят дела с попытками чтения информации в место, на которое указывает `ps`. Поскольку указатель `ps` не инициализирован, вы не можете знать, куда попадет введенная информация. Она может даже перезаписать ту информацию, которая уже имеется в памяти. К счастью, такой проблемы легко избежать: вы просто применяете достаточно большой массив `char`, чтобы принять ввод, но не используете для ввода строковых констант и неинициализированных указателей. (Или же можете обойти все эти проблемы и работать вместо массивов с объектами `std::string`.)

Внимание!

При вводе строки внутри программы всегда необходимо использовать адрес ранее распределенной памяти. Этот адрес может иметь форму имени массива либо указателя, инициализированного с помощью операции `new`.

Далее обратите внимание на то, что делает следующий код:

```
ps = animal;  // установить в ps указатель на строку
...
cout << animal << " at " << (int *) animal << endl;
cout << ps << " at " << (int *) ps << endl;
```

Он генерирует следующий вывод:

```
fox at 0x0065fd30
fox at 0x0065fd30
```

Обычно если объекту `cout` передается указатель, он печатает адрес. Но если указатель имеет тип `char *`, то `cout` отображает строку, на которую установлен указатель. Если вы хотите увидеть адрес строки, для этого потребуется выполнить приведение типа к указателю на другой тип, такой как `int *`, что и делает показанный код. Поэтому `ps` отображается как строка `"fox"`, но `(int *) ps` выводится как адрес, по которому эта строка находится. Обратите внимание, что присваивание `animal` переменной `ps` не копирует строку; оно копирует только адрес. В результате два указателя (`animal` и `ps`) указывают на одно и то же место в памяти — т.е. на одну строку.

Чтобы получить копию строки, потребуется сделать кое-что еще. Первый подход предусматривает распределение памяти для хранения копии строки. Это можно сделать либо за счет объявления еще одного массива, либо с помощью операции `new`. Второй подход позволяет точно настроить размер хранилища для строки:

```
ps = new char[strlen(animal) + 1];  // получить новое хранилище
```

Строка `"fox"` не полностью заполняет массив `animal`, поэтому при таком подходе память расходуется непроизводительно. Здесь же мы видим использование `strlen()`

для нахождения длины строки, а затем к найденной длине прибавляется единица, чтобы получить длину, включающую нулевой символ. Далее программа применяет new для выделения достаточного пространства под хранение строки.

Вам необходим способ копирования строки из массива animal во вновь выделенное пространство. Установка ps в animal не работает, потому что это только изменяет адрес, сохраненный в ps, при этом утрачивается единственная возможность доступа к выделенной памяти. Поэтому взамен необходимо применять библиотечную функцию strcpy():

```
strcpy(ps, animal);            // скопировать строку в новое хранилище
```

Функция strcpy() принимает два аргумента. Первый представляет собой целевой адрес, а второй — адрес строки, которую следует скопировать. Ваша обязанность — обеспечить, чтобы место назначения действительно смогло вместить копируемую строку. Здесь это достигается использованием функции strlen() для определения корректного размера и применением операции new для получения свободной памяти.

Обратите внимание, что за счет использования strlen() и new получены две отдельных копии "fox":

```
fox at 0x0065fd30
fox at 0x004301c8
```

Также обратите внимание, что новое хранилище располагается в памяти довольно далеко от того места, где хранится содержимое массива animal.

Вам часто придется сталкиваться с необходимостью размещения строки в массиве. Для этого можно воспользоваться операцией = при инициализации массива; иначе придется иметь дело с функцией strcpy() или strncpy(). Вы уже видели функцию strcpy(); она работает следующим образом:

```
char food[20] = "carrots";     // инициализация
strcpy(food, "flan");          // альтернатива
```

Обратите внимание, что следующий подход может послужить причиной проблем, если массив food окажется меньше, чем строка:

```
strcpy(food, "a picnic basket filled with many goodies");
```

В этом случае функция копирует остаток строки в байты памяти, непосредственно следующие за массивом, при этом перезаписывая ее содержимое, несмотря на то, что, возможно, программа ее использует для других целей. Чтобы избежать такой проблемы, вместо strcpy() вы должны применять strncpy(). Эта функция принимает третий аргумент — максимальное количество копируемых символов. Однако при этом имейте в виду, что если данная функция исчерпает свободное пространство еще до достижения конца строки, то нулевой символ она не добавит. Потому применять ее нужно так:

```
strncpy(food, "a picnic basket filled with many goodies", 19);
food[19] = '\0';
```

Этот код копирует до 19 символов в массив, после чего устанавливает последний элемент массива в нулевой символ. Если строка короче, чем 19 символов, то strncpy() добавит нулевой символ ранее, пометив им действительный конец строки.

На заметку!

Для копирования строки в массив применяйте strcpy() или strncpy(), а не операцию присваивания.

Теперь, когда вы ознакомились с некоторыми аспектами использования строк в стиле C и библиотеки cstring, вы сможете по достоинству оценить сравнительную простоту работы с типом string в C++. Обычно вам не следует беспокоиться о переполнении массива строкой, и вы можете применять операцию присваивания вместо strcpy() или strncpy().

Использование операции new для создания динамических структур

Вы уже видели, насколько выгодным может быть создание массивов во время выполнения по сравнению с этапом компиляции. То же самое касается структур. Вам нужно выделить пространство для стольких структур, сколько понадобится программе при каждом конкретном запуске. Инструментом для этого, опять-таки, послужит операция new. С ее помощью можно создавать динамические структуры. *Динамические* здесь снова означает выделение памяти во время выполнения, а не во время компиляции. Кстати, поскольку классы очень похожи на структуры, вы сможете использовать изученные приемы как для структур, так и для классов.

Применение new со структурами состоит из двух частей: создание структуры и обращение к ее членам. Для создания структуры вместе с операцией new указывается тип структуры. Например, чтобы создать безымянную структуру типа inflatable и присвоить ее адрес соответствующему указателю, можно поступить следующим образом:

```
inflatable * ps = new inflatable;
```

Это присвоит указателю ps адрес участка памяти достаточного размера, чтобы вместить тип inflatable. Обратите внимание, что синтаксис в точности такой же, как и для встроенных типов C++.

Более сложная часть — доступ к членам. Когда вы создаете динамическую структуру, то не можете применить операцию членства к имени структуры, поскольку эта структура безымянна. Все, что у вас есть — ее адрес. В C++ предусмотрена специальная операция для этой ситуации — операция членства через указатель (->). Эта операция, оформленная в виде тире с последующим значком "больше", означает для указателей на структуры то же самое, что операция точки для имен структур. Например, если ps указывает на структуру типа inflatable, то ps->price означает член price структуры, на которую указывает ps (рис. 4.11).

> **Совет**
>
> Иногда новички в C++ путаются в том, когда при обращении к членам структуры нужно применять операцию ., а когда — операцию ->. Правило простое: если идентификатор структуры представляет собой имя структуры, используйте операцию принадлежности; если же идентификатор является указателем на структуру, применяйте операцию членства через указатель.

Существует еще один, довольно неуклюжий подход для обращения к членам структур; он принимает во внимание, что если ps — указатель на структуру, то *ps — сама структура. Поэтому, если ps — структура, то (*ps).price — ее член price. Правила приоритетов операций C++ требуют применения скобок в этой конструкции.

В листинге 4.21 используется операция new для создания неименованной структуры, а также демонстрируются оба варианта нотации для доступа к ее членам.

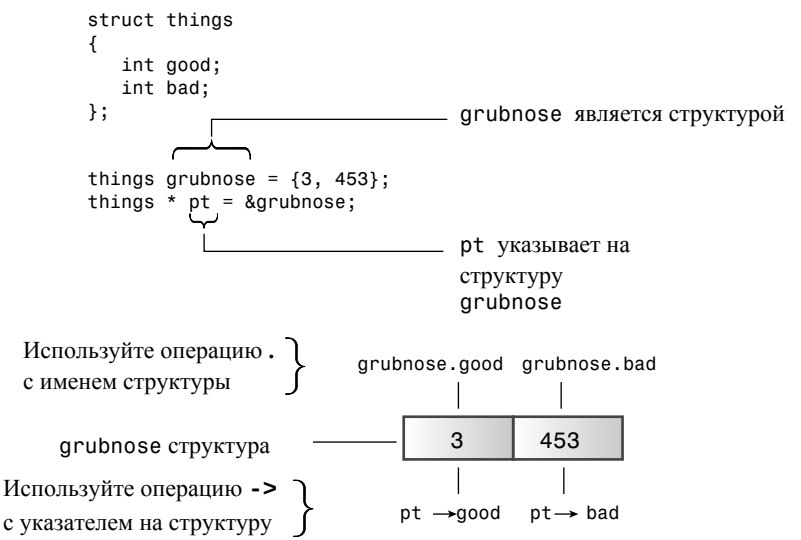


***Рис. 4.11.** Идентификация членов структуры*

Листинг 4.21. newstrct.cpp

```
// newstrct.cpp -- использование new со структурой
#include <iostream>
struct inflatable   // определение структуры
{
   char name[20];
   float volume;
   double price;
};
int main()
{
   using namespace std;
   inflatable * ps = new inflatable;         // выделение памяти для структуры
   cout << "Enter name of inflatable item: ";// ввод имени элемента inflatable
   cin.get(ps->name, 20);                    // первый метод для доступа к членам
   cout << "Enter volume in cubic feet: ";   // ввод объема в кубических футах
   cin >> (*ps).volume;                      // второй метод для доступа к членам
   cout << "Enter price: $";                 // ввод цены
   cin >> ps->price;
   cout << "Name: " << (*ps).name << endl;                  // второй метод
   cout << "Volume: " << ps->volume << " cubic feet\n";     // первый метод
   cout << "Price: $" << ps->price << endl;                 // первый метод
   delete ps;                    // освобождение памяти, использованной структурой
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 4.21:

```
Enter name of inflatable item: Fabulous Frodo
Enter volume in cubic feet: 1.4
Enter price: $27.99
Name: Fabulous Frodo
Volume: 1.4 cubic feet
Price: $27.99
```

Пример использования операций new и delete

Давайте рассмотрим пример, в котором используются операции new и delete для управления сохранением строкового ввода с клавиатуры. В листинге 4.22 определяется функция getname(), которая возвращает указатель на входную строку. Эта функция читает ввод в большой временный массив, а затем использует new[] с указанием соответствующего размера, чтобы выделить фрагмент памяти в точности такого размера, который позволит вместить входную строку. После этого функция возвращает указатель на этот блок. Такой подход может сэкономить огромный объем памяти в программе, читающей большое количество строк. (В реальности было бы проще воспользоваться классом string, в котором операции new и delete применяются внутренне.)

Предположим, что ваша программа должна прочитать 1000 строк, самая длинная из которых может составлять 79 символов, но большинство строк значительно короче. Если вы решите использовать массивы char для хранения строк, то вам понадобится 1000 массивов по 80 символов каждый, т.е. 80 000 байт, причем большая часть этого блока памяти останется неиспользованной. В качестве альтернативы можно создать массив из 1000 указателей на char и применить new для выделения ровно такого объема памяти, сколько необходимо для каждой строки. Это может сэкономить десятки тысяч байт. Вместо того чтобы создавать большой массив для каждой строки, вы выделяете память, достаточную для размещения ввода. И более того, вы можете использовать new для выделения памяти лишь для стольких указателей, сколько будет входных строк в действительности. Да, это несколько амбициозно для начала. Даже массив из 1000 указателей — довольно амбициозное решение для настоящего момента, но в листинге 4.22 демонстрируется этот прием. К тому же, чтобы проиллюстрировать работу операции delete, программа использует ее для освобождения выделенной памяти.

Листинг 4.22. delete.cpp

```
// delete.cpp -- использование операции delete
#include <iostream>
#include <cstring>              // или string.h
using namespace std;
char * getname(void);           // прототип функции
int main()
{
    char * name;                // создание указателя, но без хранилища
    name = getname();           // присваивание name адреса строки
    cout << name << " at " << (int *) name << "\n";
    delete [] name;             // освобождение памяти
    name = getname();           // повторное использование освобожденной памяти
    cout << name << " at " << (int *) name << "\n";
    delete [] name;             // снова освобождение памяти
    return 0;
}
char * getname()                // возвращает указатель на новую строку
{
    char temp[80];              // временное хранилище
    cout << "Enter last name:"; // ввод фамилии
    cin >> temp;
    char * pn = new char[strlen(temp) + 1];
    strcpy(pn, temp);           // копирование строки в меньшее пространство
    return pn;                  // по завершении функции temp теряется
}
```

Ниже приведен пример выполнения программы из листинга 4.22:

```
Enter last name: Fredeldumpkin
Fredeldumpkin at 0x004326b8
Enter last name: Pook
Pook at 0x004301c8
```

Замечания по программе

Рассмотрим функцию `getname()`, представленную в листинге 4.22. Она использует `cin` для размещения введенного слова в массив `temp`. Далее она обращается к `new` для выделения памяти, достаточной, чтобы вместить это слово. С учетом нулевого символа программе требуется сохранить в строке `strlen(temp) + 1` символов, поэтому именно это значение передается `new`. После получения пространства памяти `getname()` вызывает стандартную библиотечную функцию `strcpy()`, чтобы скопировать строку `temp` в выделенный блок памяти. Функция не проверяет, поместится ли строка, но `getname()` гарантирует выделение блока памяти подходящего размера. В конце функция возвращает `ps` — адрес копии строки.

Внутри `main()` возвращенное значение (адрес) присваивается указателю `name`. Этот указатель объявлен в `main()`, но указывает на блок памяти, выделенный в функции `getname()`. Затем программа печатает строку и ее адрес.

Далее, после освобождения блока, на который указывает `name`, функция `main()` вызывает `getname()` второй раз. C++ не гарантирует, что только что освобожденная память будет выделена при следующем вызове `new`, и, как видно из вывода программы, это и не происходит.

Обратите внимание, что в рассматриваемом примере `getname()` выделяет память, а `main()` освобождает ее. Обычно это не слишком хорошая идея — размещать `new` и `delete` в разных функциях, потому что в таком случае очень легко забыть вызвать `delete`. В этом примере мы разделили эти две операции просто для того, чтобы продемонстрировать, что подобное возможно.

Благодаря некоторым тонким аспектам данной программы, вы должны узнать немного больше о том, как C++ управляет памятью. Поэтому давайте предварительно ознакомимся с материалом, который будет раскрыт более подробно в главе 9.

Автоматическое, статическое и динамическое хранилище

В C++ предлагаются три способа управления памятью для данных — в зависимости от метода ее выделения: автоматическое хранилище, статическое хранилище и динамическое хранилище, иногда называемое *свободным хранилищем* или *кучей*. Объекты данных, выделенные этими тремя способами, отличаются друг от друга тем, насколько долго они существуют. Рассмотрим кратко каждый из них. (В C++11 добавляется четвертая форма, которая называется *хранилищем потока* и кратко рассматривается в главе 9.)

Автоматическое хранилище

Обычные переменные, объявленные внутри функции, используют *автоматическое хранилище* и называются *автоматическими переменными*. Этот термин означает, что они создаются автоматически при вызове содержащей их функции и уничтожаются при ее завершении. Например, массив `temp` в листинге 4.22 существует только во время работы функции `getname()`. Когда управление программой возвращается `main()`, то память, используемая `temp`, освобождается автоматически. Если бы `getname()` возвращала указатель на `temp`, то указатель `name` в `main()` остался бы установленным на

адрес памяти, которая скоро может быть использована повторно. Вот почему внутри `getname()` необходимо вызывать `new`. На самом деле автоматические значения являются локальными по отношению к блоку, в котором они объявлены. *Блок* — это раздел кода, ограниченный фигурными скобками. До сих пор все наши блоки были целыми функциями. Но как будет показано в следующей главе, можно иметь блоки и внутри функций. Если вы объявите переменную внутри одного из таких блоков, она будет существовать только в то время, когда программа выполняет операторы, содержащиеся внутри этого блока.

Автоматические переменные обычно хранятся в *стеке*. Это значит, что когда выполнение программы входит в блок кода, его переменные последовательно добавляются к стеку в памяти и затем освобождаются в обратном порядке, когда выполнение покидает данный блок. (Этот процесс называется *LIFO* (last-in, first-out — "последним пришел — первым ушел".) Таким образом, по мере продвижения выполнения стек растет и уменьшается.

Статическое хранилище

Статическое хранилище — это хранилище, которое существует в течение всего времени выполнения программы. Доступны два способа для того, чтобы сделать переменные статическими. Один заключается в объявлении их вне функций. Другой предполагает использование при объявлении переменной ключевого слова `static`:

```
static double fee = 56.50;
```

Согласно правилам языка C стандарта K&R, вы можете инициализировать только статические массивы и структуры, в то время как C++ Release 2.0 (и более поздние), а также ANSI C позволяют также инициализировать автоматические массивы и структуры. Однако, как вы, возможно, обнаружите, в некоторых реализациях C++ до сих пор не реализована инициализация таких массивов и структур.

В главе 9 статическое хранилище обсуждается более подробно. Главный момент, который вы должны запомнить сейчас относительно автоматического и статического хранилищ — то, что эти методы строго определяют время жизни переменных. Переменные могут либо существовать на протяжении всего выполнения программы (статические переменные), либо только в период выполнения функции или блока (автоматические переменные).

Динамическое хранилище

Операции `new` и `delete` предлагают более гибкий подход, нежели использование автоматических и статических переменных. Они управляют пулом памяти, который в C++ называется *свободным хранилищем* или *кучей*. Этот пул отделен от области памяти, используемой статическими и автоматическими переменными. Как было показано в листинге 4.22, операции `new` и `delete` позволяют выделять память в одной функции и освобождать в другой. Таким образом, время жизни данных при этом не привязывается жестко к времени жизни программы или функции.

Совместное применение `new` и `delete` предоставляет возможность более тонко управлять использованием памяти, чем в случае обычных переменных. Однако управление памятью становится более сложным. В стеке механизм автоматического добавления и удаления приводит к тому, что части памяти всегда являются смежными. Тем не менее, чередование операций `new` и `delete` может привести к появлению промежутков в свободном хранилище, усложняя отслеживание места, где будут распределяться новые запросы памяти.

Стеки, кучи и утечка памяти

Что произойдет, если не вызвать `delete` после создания переменной с помощью операции `new` в свободном хранилище (куче)? Если не вызвать `delete`, то переменная или конструкция, динамически выделенная в области свободного хранилища, останется там, даже при условии, что память, содержащая указатель, будет освобождена в соответствии с правилами видимости и временем жизни объекта. По сути, после этого у вас не будет никакой возможности получить доступ к такой конструкции, находящейся в области свободного хранилища, поскольку уже не будет существовать указателя, который помнит ее адрес. В этом случае вы получите *утечку памяти*. Такая память остается недоступной на протяжении всего сеанса работы программы. Она была выделена, но не может быть освобождена. В крайних случаях (хотя и нечастых), утечки памяти могут привести к тому, что они поглотят всю память, выделенную программе, что вызовет ее аварийное завершение с сообщением об ошибке переполнения памяти. Вдобавок такие утечки могут негативно повлиять на некоторые операционные системы и другие приложения, использующие то же самое пространство памяти, приводя к их сбоям.

Даже лучшие программисты и программистские компании допускают утечки памяти. Чтобы избежать их, лучше выработать привычку сразу объединять операции `new` и `delete`, тщательно планируя создание и удаление конструкций, как только вы собираетесь обратиться к динамической памяти. Помочь автоматизировать эту задачу могут интеллектуальные указатели C++ (см. главу 16).

На заметку!

Указатели — одно из наиболее мощных средств C++. Однако они также и наиболее опасны, потому что открывают возможность недружественных к компьютеру действий, таких как использование неинициализированных указателей для доступа к памяти либо попыток освобождения одного и того же блока дважды. Более того, до тех пор, пока вы не привыкнете в нотации указателей и к самой концепции указателей на практике, они будут приводить к путанице. Но поскольку указатели — важнейшая часть программирования на C++, они постоянно будут присутствовать во всех дальнейших обсуждениях. К теме указателей мы еще будем обращаться не раз. Мы надеемся, что каждое объяснение поможет вам чувствовать себя все более уверенно при работе с указателями.

Комбинации типов

В этой главе были представлены массивы, структуры и указатели. Они могут комбинироваться разнообразными способами, поэтому давайте рассмотрим некоторые возможности, начав со структуры:

```
struct antarctica_years_end
{
    int year;
    /* определение других нужных данных */
};
```

Можно создавать переменные этого типа:

```
antarctica_years_end s01, s02, s03;   // s01, s02, s03 - структуры
```

После этого можно обращаться к членам с использованием операции принадлежности:

```
s01.year = 1998;
```

Можно создать указатель на такую структуру:

```
antarctica_years_end * pa = &s02;
```

Имея указатель, установленный в допустимый адрес, можно использовать операцию членства через указатель для доступа к членам:

```
pa->year = 1999;
```

Можно создавать массивы структур:

```
antarctica_years_end trio[3];   // массив из трех структур
```

Затем с помощью операции принадлежности можно обращаться к членам какого-нибудь элемента:

```
trio[0].year = 2003;            // trio[0] является структурой
```

Здесь `trio` — это массив, но `trio[0]` — структура, и `trio[0].year` представляет собой член этой структуры. Поскольку имя массива является указателем, можно также применять операцию членства через указатель:

```
(trio+1)->year = 2004;          // то же, что и trio[1].year = 2004;
```

Можно создавать массивы указателей:

```
const antarctica_years_end * arp[3] = {&s01, &s02, &s03};
```

Это уже выглядит немного сложнее. Как получить доступ к данным в этом массиве? Если `arp` — массив указателей, то `arp[1]` должен быть указателем, и для доступа к члену можно воспользоваться операцией членства через указатель:

```
std::cout << arp[1]->year << std::endl;
```

Можно создавать указатель на такой массив:

```
const antarctica_years_end ** ppa = arp;
```

Здесь `arp` представляет собой имя массива; следовательно, он является адресом первого элемента в массиве. Но его первый элемент — указатель, поэтому `ppa` должен быть указателем на указатель на `const antarctica_years_end`, отсюда и `**`. Существует немало путей внести путаницу в это объявление. Например, можно было бы забыть о `const` или о звездочке (а то и о двух), переставить буквы либо еще каким-то образом исказить этот тип структуры. Это может служить примером удобства применения ключевого слова `auto` из C++11. Компилятор хорошо осведомлен о типе `arp`, поэтому он может вывести правильный тип самостоятельно:

```
auto ppb = arp;                 // автоматическое выведение типа в C++11
```

В прошлом компилятор использовал свои знания о правильном типе для сообщения об ошибках, которые вы могли допустить в объявлении; теперь же эти знания работают на вас.

Как использовать `ppa` для доступа к данным? Поскольку `ppa` — это указатель на указатель на структуру, `*ppa` представляет собой указатель на структуру, так что его можно применять с операцией членства через указатель:

```
std::cout << (*ppa)->year << std::endl;
std::cout << (*(ppb+1))->year << std::endl;
```

Так как `ppa` указывает на первый член `arp`, `*ppa` является первым членом, т.е. `&s01`. Таким образом, `(*ppa)->year` — это член `year` в `s01`. Во втором операторе `ppb+1` указывает на следующий элемент, `arp[1]`, т.е. `&s02`. Круглые скобки нужны для обеспечения корректной ассоциации. Например, `*ppa->year` будет пытаться применить операцию `*` к `ppa->year`, что даст сбой, поскольку член `year` не является указателем.

Действительно ли это все так? В листинге 4.23 все предшествующие операторы скомбинированы в одну короткую программу.

Листинг 4.23. mixtypes.cpp

```
// mixtypes.cpp -- некоторые комбинации типов
#include <iostream>
struct antarctica_years_end
{
    int year;
    /* определение других нужных данных */
};

int main()
{
    antarctica_years_end s01, s02, s03;
    s01.year = 1998;
    antarctica_years_end * pa = &s02;
    pa->year = 1999;
    antarctica_years_end trio[3];  // массив из трех структур
    trio[0].year = 2003;
    std::cout << trio->year << std::endl;
    const antarctica_years_end * arp[3] = {&s01, &s02, &s03};
    std::cout << arp[1]->year << std::endl;
    const antarctica_years_end ** ppa = arp;
    auto ppb = arp;                  // автоматическое выведение типа в C++11

    // или можно использовать const antarctica_years_end ** ppb = arp;
    std::cout << (*ppa)->year << std::endl;
    std::cout << (*(ppb+1))->year << std::endl;
    return 0;
}
```

Ниже показан вывод:

```
2003
1999
1998
1999
```

Программа компилируется и работает так, как ожидалось.

Альтернативы массивам

Ранее в этой главе упоминались шаблонные классы `vector` и `array` как альтернативы встроенному массиву. Давайте кратко рассмотрим, как их использовать и какие это может принести выгоды.

Шаблонный класс `vector`

Шаблонный класс `vector` похож на класс `string` в том, что он является динамическим массивом. Установить размер объекта `vector` можно во время выполнения, и можно добавлять новые данные в конец или вставлять их в середину. В основном `vector` представляет собой альтернативу применению операции `new` для создания динамического массива. На самом деле класс `vector` использует операции `new` и `delete` для управления памятью, но делает это автоматически.

На данный момент мы не планируем глубоко погружаться в то, что собой представляет шаблонный класса. Вместо этого мы рассмотрим несколько базовых практических вопросов. Во-первых, чтобы можно было работать с объектом vector, понадобится включить заголовочный файл vector. Во-вторых, идентификатор vector является частью пространства имен std, поэтому придется использовать директиву using, объявление using или запись std::vector. В-третьих, шаблоны применяют другой синтаксис для указания типа сохраненных данных. В-четвертых, класс vector использует отличающийся синтаксис для указания количества элементов. Ниже показаны некоторые примеры:

```
#include <vector>
...
using namespace std;
vector<int> vi;          // создание массива int нулевого размера
int n;
cin >> n;
vector<double> vd(n);    // создание массива из n элементов double
```

На основе этого кода можно сказать, что vi — это объект типа vector<int>, а vd — объект типа vector<double>. Поскольку объекты vector изменяют свои размеры автоматически при вставке или добавлении значений к ним, вполне нормально для vi начать с размера 0. Но чтобы изменение размера работало, необходимо применять разнообразные методы, входящие в состав пакета vector.

В общем, следующее объявление создает объект vector по имени vt, который может хранить *количество_элементов* элементов типа *имяТипа*:

```
vector<имяТипа> vt(количество_элементов);
```

Параметр *количество_элементов* может быть целочисленной константой или целочисленной переменной.

Шаблонный класс array (C++11)

Класс vector обладает бо́льшими возможностями, чем встроенный тип массива, но достигается это ценой некоторого снижения эффективности. Если все, что требуется — это массив фиксированного размера, может быть выгоднее использовать встроенный тип. Однако при этом снижается степень удобства и безопасности. C++11 реагирует на эту ситуацию добавлением шаблонного класса array, который является частью пространства имен std. Подобно встроенному типу, объект array имеет фиксированный размер и использует стек (или распределение в статической памяти) вместо свободного хранилища, поэтому он характеризуется эффективностью встроенных массивов. К этому добавляется удобство и безопасность. Для создания объекта array должен быть включен заголовочный файл array. Используемый синтаксис несколько отличается от такового для vector:

```
#include <array>
...
using namespace std;
array<int, 5> ai;        // создание объекта array из пяти элементов int
array<double, 4> ad = {1.2, 2.1, 3.43. 4.3};
```

В общем случае следующее объявление создает объект array по имени arr, который может хранить *количество_элементов* элементов типа *имяТипа*:

```
array<имяТипа, количество_элементов> arr;
```

В отличие от vector, *количество_элементов* не может быть переменной.

В C++11 можно применять списковую инициализацию для объектов vector и array. Тем не менее, это не доступно для объектов vector в C++98.

Сравнение массивов, объектов vector и объектов array

Проще всего понять сходства и различия между массивами, объектами vector и объектами array, рассмотрев краткий пример (листинг 4.24), в котором используются все три подхода.

Листинг 4.24. choices.cpp

```
// choices.cpp -- вариации массивов
#include <iostream>
#include <vector>        // STL C++98
#include <array>         // C++11
int main()
{
   using namespace std;

   // C, исходный C++
   double a1[4] = {1.2, 2.4, 3.6, 4.8};

   // C++98 STL
   vector<double> a2(4); // создание объекта vector с четырьмя элементами

   // Простой способ инициализации в C98 отсутствует
   a2[0] = 1.0/3.0;
   a2[1] = 1.0/5.0;
   a2[2] = 1.0/7.0;
   a2[3] = 1.0/9.0;

   // C++11 -- создание и инициализация объекта array
   array<double, 4> a3 = {3.14, 2.72, 1.62, 1.41};
   array<double, 4> a4;
   a4 = a3;              // допускается для объектов array одного и того же размера

   // Использование нотации массивов
   cout << "a1[2]: " << a1[2] << " at " << &a1[2] << endl;
   cout << "a2[2]: " << a2[2] << " at " << &a2[2] << endl;
   cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;
   cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;

   // Преднамеренная ошибка
   a1[-2] = 20.2;
   cout << "a1[-2]: " << a1[-2] <<" at " << &a1[-2] << endl;
   cout << "a3[2]: " << a3[2] << " at " << &a3[2] << endl;
   cout << "a4[2]: " << a4[2] << " at " << &a4[2] << endl;
   return 0;
}
```

Ниже показан пример вывода:

```
a1[2]: 3.6 at 0x28cce8
a2[2]: 0.142857 at 0xca0328
a3[2]: 1.62 at 0x28ccc8
a4[2]: 1.62 at 0x28cca8
a1[-2]: 20.2 at 0x28ccc8
a3[2]: 20.2 at 0x28ccc8
a4[2]: 1.62 at 0x28cca8
```

Замечания по программе

Во-первых, обратите внимание, что независимо от применяемого подхода — встроенного массива, объекта vector или объекта array — мы можем использовать стандартную нотацию массивов для доступа к индивидуальным членам. Во-вторых, по адресам легко заметить, что объекты array находятся в той же самой области памяти (в данном случае — в стеке), что и встроенный массив, тогда как объект vector хранится в другой области (в свободном хранилище, или куче). В-третьих, в коде показано, что один объект array можно присвоить другому объекту array. В случае встроенных объектов понадобится поэлементно копировать данные.

Далее обратите особое внимание на следующую строку:

```
a1[-2] = 20.2;
```

Что означает индекс -2? Вспомните, что эта запись транслируется в такой код:

```
*(a1-2) = 20.2;
```

Выразить словами это можно так: посмотреть, на что указывает a1, переместиться на два элемента double назад и поместить туда значение 20.2. То есть сохранить информацию в позиции за пределами массива. В этом конкретном случае данной позицией оказывается объект array по имени a3. Другой компилятор поместит 20.2 в a4, а прочие могут предпринять еще какие-нибудь неверные действия. Это пример небезопасного поведения встроенных массивов.

Защищают ли объекты vector и array от такого поведения? Да, они могут, если вы им позволите. То есть вы по-прежнему можете писать небезопасный код вроде такого:

```
a2[-2] = .5;          // по-прежнему разрешено
a3[200] = 1.4;
```

Однако существуют альтернативы. Одна из них предполагает применение функции-члена at(). Точно так же, как вы можете использовать функцию-член getline() с объектом cin, вы можете применять функцию-член at() с объектами vector и array:

```
a2.at(1) = 2.3;       // присваивает a2[1] значение 2.3
```

Отличие между использованием нотации с квадратными скобками и вызовом функции-члена at() состоит в том, что в случае at() указание недопустимого индекса во время выполнения по умолчанию приводит к аварийному завершению программы. За счет такой дополнительной проверки увеличивается время выполнения; именно поэтому в C++ доступен на выбор один из этих вариантов. Более того, классы vector и array предлагают способы использования объектов, которые снижают вероятность появления непредвиденных ошибок диапазона. Например, эти классы имеют функции-члены begin() и end(), позволяющие установить границы диапазона без случайного выхода за их пределы. Все это будет подробно обсуждаться в главе 16.

Резюме

Массивы, структуры и указатели являются тремя составными типами в C++. Массив может содержать множество значений одного и того же типа в единственном объекте данных. Доступ к индивидуальным элементам массива осуществляется с использованием индекса.

Структура может содержать несколько значений разных типов в одном объекте данных, и для доступа к ним можно применять операцию принадлежности или членства (.). Первым шагом при работе со структурой является создание шаблона структуры, который определяет входящие в нее члены. Имя, или дескриптор, такого шаблона затем становится идентификатором нового типа данных. После этого можно объявлять структурные переменные этого типа.

Объединение может содержать одно значение, но различных типов, причем имя члена при этом указывает, какой режим (тип) используется в конкретный момент.

Указатели — это переменные, которые предназначены для хранения адресов памяти. Говорят, что указатель указывает на адрес, который он хранит. Объявление указателя всегда включает тип объекта, на который он указывает. Применяя операцию разыменования (*), можно получить значение, находящееся в памяти по адресу, который хранится в указателе.

Строка — это последовательность символов, ограниченная нулевым символом. Строка может быть представлена в виде строковой константы, заключенной в кавычки, при этом наличие нулевого символа подразумевается неявно. Строку можно сохранить в массиве char и строку можно представить как указатель на char, инициализированный для указания на эту строку. Функция strlen() возвращает длину строки, не считая нулевой символ-ограничитель. Функция strcpy() копирует строку из одного места в другое. Для применения этих функций необходимо включить в программу заголовочный файл cstring или string.h.

Класс C++ по имени string, поддерживаемый заголовочным файлом string, предлагает альтернативный, более дружественный к пользователю способ обращения со строками. В частности, объекты string автоматически изменяют свои размеры, приспосабливаясь к хранимым строкам, а для их копирования можно применять операцию присваивания.

Операция new позволяет запрашивать память для объектов данных во время выполнения программы. Эта операция возвращает адрес выделенного участка памяти, который можно присвоить указателю. Единственный способ доступа к такой памяти — через указатель. Если объектом данных является простая переменная, вы можете применить операцию разыменования (*) для получения значения. Если объект данных — массив, вы можете использовать указатель на него как обычное имя массива и обращаться к его элементам по индексу. Если же объект данных — структура, вы можете использовать операцию -> для доступа к ее членам.

Указатели и массивы тесно связаны. Если ar — имя массива, то выражение ar[i] интерпретируется как *(ar + i), т.е. имя массива интерпретируется как адрес его первого элемента. Таким образом, имя массива играет ту же роль, что и указатель. В свою очередь, вы можете использовать имя указателя в нотации массивов, чтобы обращаться к элементам в массиве, распределенном операцией new.

Операции new и delete позволяют явно управлять тем, когда объекты данных размещаются и когда покидают пул свободной памяти. Автоматические переменные, которые объявляются внутри функций, и статические переменные, определяемые вне функций или же с помощью ключевого слова static, являются менее гибкими. Автоматические переменные создаются, когда управление приходит в содержащий их блок (обычно в теле функции), и исчезают, когда оно его покидает. Статические переменные сохраняются в течение всего времени выполнения программы.

Стандартная библиотека шаблонов (STL), добавленная стандартом C++98, предоставляет шаблонный класс vector, являющийся альтернативой самостоятельно создаваемым динамическим массивам. C++11 предлагает шаблонный класс array, который представляет собой альтернативу встроенным массивам фиксированных размеров.

Вопросы для самоконтроля

1. Как вы объявите следующие объекты данных?

   а. `actor` — массив из 30 элементов `char`.

   б. `betsie` — массив из 100 элементов `short`.

   в. `chuck` — массив из 13 элементов `float`.

   г. `dipsea` — массив из 64 элементов `long double`.

2. Выполните задание из вопроса 1, используя шаблонный класс `array` вместо встроенных массивов.

3. Объявите массив из пяти элементов `int` и инициализируйте его первыми пятью положительными нечетными числами.

4. Напишите оператор, который присваивает переменной `even` сумму первого и последнего элементов массива из вопроса 3.

5. Напишите оператор, который отображает значение второго элемента массива `float` по имени `ideas`.

6. Объявите массив `char` и инициализируйте его строкой `"cheeseburger"`.

7. Объявите объект `string` и инициализируйте его строкой `"Waldorf Salad"`.

8. Разработайте объявление структуры, описывающей рыбу. Структура должна включать вид, вес в полных унциях и длину в дробных дюймах.

9. Объявите переменную типа, определенного в вопросе 8, и инициализируйте ее.

10. Воспользуйтесь `enum` для определения типа по имени `Response` с возможными значениями `Yes`, `No` и `Maybe`. `Yes` должно быть равно `1`, `No` — `0`, а `Maybe` — `2`.

11. Предположим, что `ted` — переменная типа `double`. Объявите указатель, указывающий на `ted`, и воспользуйтесь им, чтобы отобразить значение `ted`.

12. Предположим, что `treacle` — массив из 10 элементов `float`. Объявите указатель, указывающий на первый элемент `treacle`, и используйте его для отображения первого и последнего элементов массива.

13. Напишите фрагмент кода, который запрашивает у пользователя положительное целое число и затем создает динамический массив с указанным количеством элементов типа `int`. Сделайте это с применением операции `new`, а затем с использованием объекта `vector`.

14. Правильный ли код приведен ниже? Если да, что он напечатает?

```
cout << (int *) "Home of the jolly bytes";
```

15. Напишите фрагмент кода, который динамически выделит память для структуры, описанной в вопросе 8, и затем прочитает в нее значение для члена `kind` структуры.

16. В листинге 4.6 иллюстрируется проблема, вызванная тем, что числовой ввод следует за строчно-ориентированным вводом. Как замена оператора

```
cin.getline(address,80);
```

оператором

```
cin >> address;
```

повлияет на работу этой программы?

17. Объявите объект vector из 10 объектов string и объект array из 10 объектов string. Покажите необходимые заголовочные файлы и не используйте using. Для количества строк применяйте const.

Упражнения по программированию

1. Напишите программу C++, которая запрашивает и отображает информацию, как показано в следующем примере вывода:

```
What is your first name? Betty Sue
What is your last name? Yewe
What letter grade do you deserve? B
What is your age? 22
Name: Yewe, Betty Sue
Grade: C
Age: 22
```

   Обратите внимание, что программа должна принимать имена, состоящие из более чем одного слова. Кроме того, программа должна уменьшать значение grade на одну градацию — т.е. на одну букву выше. Предполагается, что пользователь может ввести A, B или C, поэтому вам не нужно беспокоиться о пропуске между D и F.

2. Перепишите листинг 4.4, применив класс C++ string вместо массивов char.

3. Напишите программу, которая запрашивает у пользователя имя, фамилию, а затем конструирует, сохраняет и отображает третью строку, состоящую из фамилии пользователя, за которой следует запятая, пробел и его имя. Используйте массивы char и функции из заголовочного файла cstring. Пример запуска должен выглядеть так:

```
Enter your first name: Flip
Enter your last name: Fleming
Here's the information in a single string: Fleming, Flip
```

4. Напишите программу, которая приглашает пользователя ввести его имя и фамилию, а затем построит, сохранит и отобразит третью строку, состоящую из фамилии, за которой следует запятая, пробел и имя. Используйте объекты string и методы из заголовочного файла string. Пример запуска должен выглядеть так:

```
Enter your first name: Flip
Enter your last name: Fleming
Here's the information in a single string: Fleming, Flip
```

5. Структура CandyBar содержит три члена. Первый из них хранит название коробки конфет. Второй — ее вес (который может иметь дробную часть), а третий — число калорий (целое значение). Напишите программу, объявляющую эту структуру и создающую переменную типа CandyBar по имени snack, инициализируя ее члены значениями "Mocha Munch", 2.3 и 350, соответственно. Инициализация должна быть частью объявления snack. И, наконец, программа должна отобразить содержимое этой переменной.

6. Структура CandyBar включает три члена, как описано в предыдущем упражнении. Напишите программу, которая создает массив из трех структур CandyBar, инициализирует их значениями по вашему усмотрению и затем отображает содержимое каждой структуры.

7. Вильям Вингейт (William Wingate) заведует службой анализа рынка пиццы. О каждой пицце он записывает следующую информацию:
   - наименование компании — производителя пиццы, которое может состоять из более чем одного слова;
   - диаметр пиццы;
   - вес пиццы.
   - Разработайте структуру, которая может содержать всю эту информацию, и напишите программу, использующую структурную переменную этого типа. Программа должна запрашивать у пользователя каждый из перечисленных показателей и затем отображать введенную информацию. Применяйте `cin` (или его методы) и `cout`.
8. Выполните упражнение 7, но с применением операции `new` для размещения структуры в свободном хранилище вместо объявления структурной переменной. Кроме того, сделайте так, чтобы программа сначала запрашивала диаметр пиццы, а потом — наименование компании.
9. Выполните упражнение 6, но вместо объявления массива из трех структур `CandyBar` используйте операцию `new` для динамического размещения массива.
10. Напишите программу, которая приглашает пользователя ввести три результата забега на 40 ярдов (или 40 метров, если желаете) и затем отображает эти значения и их среднее. Для хранения данных применяйте объект `array`. (Если объект `array` не доступен, воспользуйтесь встроенным массивом.)

5

Циклы и выражения отношений

В ЭТОЙ ГЛАВЕ...

- Цикл `for`
- Выражения и операторы
- Операции инкремента и декремента: `++` и `--`
- Комбинированные операции присваивания
- Составные операторы (блоки)
- Операция запятой
- Операции сравнения: `>`,`>=`,`==`,`<=`,`<` и `!=`
- Цикл `while`
- Средство `typedef`
- Цикл `do while`
- Метод ввода символов `get()`
- Условие конца файла
- Вложенные циклы и двумерные массивы

Компьютеры умеют намного больше, чем просто хранить данные. Они анализируют, объединяют, упорядочивают, извлекают, модифицируют, экстраполируют, синтезируют и выполняют другие манипуляции над данными. Иногда они даже искажают и уничтожают данные, но мы стараемся контролировать такое их поведение. Чтобы творить все эти чудеса, программам необходимы инструменты для выполнения повторяющихся действий и принятия решений. Конечно, язык C++ предоставляет такие инструменты. На самом деле в нем используются те же циклы `for`, `while`, `do while` и операторы `if`, `switch`, которые есть в языке C, поэтому если вы знаете C, то можете быстро пробежаться по этой и следующей главе. (Но все-таки не слишком спешите — вы же не хотите пропустить объяснение того, как объект `cin` обрабатывает символьный ввод.) Все эти разнообразные управляющие операторы часто используют выражения сравнения и логические выражения для обеспечения должного поведения программы. В настоящей главе рассматриваются циклы и выражения отношений, а в главе 6 — операторы ветвления и логические выражения.

Введение в циклы `for`

Обстоятельства часто требуют от программ выполнения повторяющихся задач, таких как сложение элементов массивов один за другим или 20-кратная распечатка похвалы за продуктивность. Цикл `for` облегчает выполнение задач подобного рода. Давайте взглянем на цикл в листинге 5.1, посмотрим, что он делает, а затем разберемся, как он работает.

Листинг 5.1. `forloop.cpp`

```
// forloop.cpp -- представление цикла for
#include <iostream>
int main()
{
    using namespace std;
    int i;  // создание счетчика
//   инициализация; проверка ; обновление
    for (i = 0; i < 5; i++)
        cout << "C++ knows loops.\n";
    cout << "C++ knows when to stop.\n";
    return 0;
}
```

Ниже показан вывод программы из листинга 5.1:

```
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows loops.
C++ knows when to stop.
```

Этот цикл начинается с присваивания целочисленной переменной `i` значения 0:

```
i = 0;
```

Это — часть *инициализации цикла*. Затем в части *проверки цикла* программа проверяет, меньше ли `i` числа 5:

```
i < 5;
```

Если это так, программа выполняет следующий оператор, который называется *телом цикла*:

```
cout << "C++ knows loops.\n";
```

После этого программа активизирует часть *обновления цикла*, увеличивая `i` на 1:

```
i++
```

В части обновления цикла используется операция ++, которая называется *операцией инкремента*. Она увеличивает значение своего операнда на 1. (Применение операции инкремента не ограничено циклами `for`. Например, можно использовать `i++;` вместо `i = i + 1;` в качестве оператора программы.) Инкрементирование `i` завершает первый проход цикла.

Далее цикл начинает новый проход, сравнивая новое значение `i` с 5. Поскольку новое значение (1) также меньше 5, цикл печатает еще одну строку и завершается снова инкрементированием `i`. Это подготавливает новый проход цикла — проверку, выполнение оператора и обновление значения `i`. Процесс продолжается до тех пор, пока не `i` не получит значение 5. После этого следующая проверка дает ложный результат, и программа переходит к оператору, следующему за циклом.

Части цикла `for`

Цикл `for` представляет собой средство пошагового выполнения повторяющихся действий. Давайте рассмотрим более подробно, как он устроен. Обычно части цикла `for` выполняют следующие шаги.

1. Установка начального значения.
2. Выполнение проверки условия для продолжения цикла.
3. Выполнение действий цикла.
4. Обновление значения (значений), используемых в проверочном условии.

В структуре цикла C++ эти элементы расположены таким образом, чтобы их можно было охватить одним взглядом. Инициализация, проверка и обновление составляют три части управляющего раздела, заключенного в круглые скобки. Каждая часть является выражением, отделяемым от других частей точкой с запятой. Оператор, следующий за управляющим разделом, называется *телом* цикла, и он выполняется до тех пор, пока проверочное условие остается истинным:

```
for (инициализация; проверочное выражение; обновляющее выражение)
    тело
```

В синтаксисе C++ полный оператор `for` считается одним оператором, несмотря на то, что он может заключать в своем теле один или более других операторов. (При наличии нескольких операторов в теле цикла они должны быть помещены в блок, как будет описано ниже.)

Цикл выполняет *инициализацию* только однажды. Как правило, программы используют это выражение для установки переменной в некоторое начальное значение, а потом применяют эту переменную в качестве счетчика цикла.

Проверочное выражение определяет, должно ли выполняться тело цикла. Обычно это выражение представляет собой выражение сравнения — т.е. выражение, сравнивающее два значения. В нашем примере сравниваются значения `i` и 5. Если проверка проходит успешно (проверочное выражение истинно), программа выполняет тело цикла. На самом деле в C++ проверочные выражения не ограничиваются толь-

ко сравнениями, дающими в результате `true` или `false`. Здесь можно использовать любое выражение, и C++ приведет его к типу `bool`. Таким образом, выражение, возвращающее `0`, преобразуется в булевское значение `false`, и цикл завершается. Если выражение оценивается как ненулевое, оно приводится к булевскому значению `true`, и цикл продолжается. В листинге 5.2 это демонстрируется на примере использования выражения `i` в качестве проверочного. (В разделе обновления `i--` подобно `i++`, за исключением того, что оно уменьшает значение `i` на 1 при каждом выполнении.)

Листинг 5.2. `num_test.cpp`

```
// num_test.cpp -- использование числовой проверки в цикле
#include <iostream>
int main()
{
  using namespace std;
  cout << "Enter the starting countdown value: "; // ввод начального значения счетчика
  int limit;
  cin >> limit;
  int i;
  for (i = limit; i; i--)                          // завершается, когда i равно 0.
     cout << "i = " << i << "\n";
  cout << "Done now that i = " << i << "\n";       // цикл завершен, вывод значения i
  return 0;
}
```

Вот как выглядит пример выполнения программы из листинга 5.2:

```
Enter the starting countdown value: 4
i = 4
i = 3
i = 2
i = 1
Done now that i = 0
```

Обратите внимание, что цикл завершается, когда `i` достигает значения 0.

Каким же образом сравнивающие выражения вроде `i < 5` вписываются в концепцию завершения цикла при достижении значения 0? До появления типа `bool` сравнивающие выражения вычислялись как `1` в случае истинности и `0` — в противоположном случае. Таким образом, значение выражения `3 < 5` было равно `1`, а значение `5 < 5` — равно `0`. Теперь, когда в C++ появился тип `bool`, сравнивающие выражения возвращают вместо `1` и `0` булевские литералы `true` и `false`. Однако это изменение не приводит к несовместимости, потому что программы C++ преобразуют `true` и `false` в `1` и `0` там, где ожидаются целые значения, а `0` преобразует в `false` и ненулевое значение — в `true` там, где ожидаются значения типа `bool`.

Цикл `for` является циклом с *входным условием*. Это значит, что проверочное условие выполняется *перед* каждым шагом цикла. Цикл никогда не выполняет тело, если проверочное условие возвращает `false`. Например, представьте, что вы запустили программу из листинга 5.2, но в качестве начального значения ввели `0`. Поскольку проверочное условие не удовлетворено при первой же проверке, тело цикла не выполнится ни разу:

```
Enter the starting countdown value: 0
Done now that i = 0
```

Позиция "проверка перед циклом" может помочь предохранить программу от проблем.

Обновляющее выражение вычисляется в конце цикла, после того, как выполнено тело цикла. Обычно оно используется для увеличения или уменьшения значения переменной, управляющей количеством шагов цикла. Однако оно может быть любым допустимым выражением C++, как и все остальные управляющие выражения. Это обеспечивает циклу `for` гораздо более широкие возможности, чем простой отсчет от 0 до 5, как делалось в первом примере. Позже вы увидите некоторые примеры этих возможностей. Тело цикла `for` состоит из одного оператора, но вскоре вы увидите, как расширить это правило. На рис. 5.1 показана блок-схема цикла `for`.

```
оператор1
for (инициализационное_выражение;
     проверочное_выражение;
     обновляющее_выражение)
  оператор2
оператор3
```

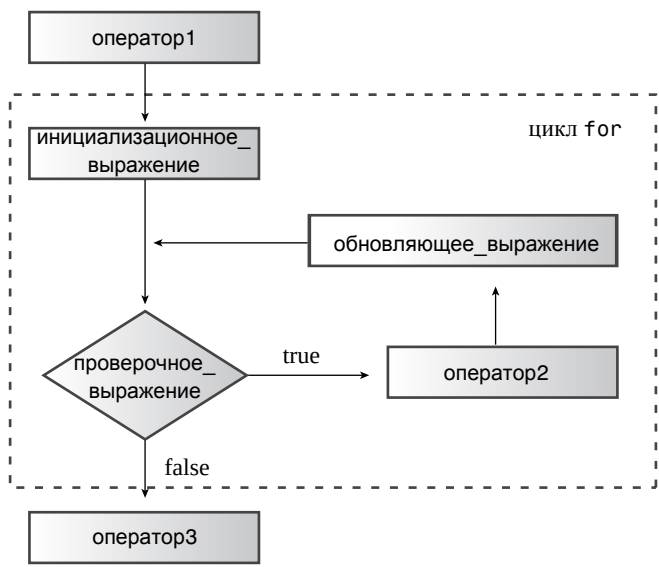


***Рис. 5.1.** Структура циклов `for`*

Оператор цикла `for` в чем-то похож на вызов функции, потому что использует имя, за которым следует пара скобок. Однако статус `for` как ключевого слова C++ предотвращает восприятие его компилятором как функции. Это также предохраняет вас от попыток назвать функцию именем `for`.

Совет

В C++ принят стиль помещения пробелов между `for` и последующими скобками, а также пропуск пробела между именем функции и следующими за ним скобками:

```
for (i = 6; i < 10; i++)
  smart_function(i);
```

Другие управляющие операторы, такие как `if` и `while`, трактуются аналогично `for`. Это служит для визуального подчеркивания разницы между управляющим оператором и вызовом функции. К тому же общепринятая практика заключается в снабжении тела функции отступом, чтобы выделить его визуально.

Выражения и операторы

Управляющий раздел `for` включает три выражения. В пределах ограничений, накладываемых его синтаксисом, C++ является очень выразительным языком. Любое значение или любая допустимая комбинация выражений и операций составляет выражение. Например, `10` — это выражение со значением 10 (что не удивительно), а `28 * 20` — выражение со значением 560. В C++ любое выражение имеет свое значение. Часто оно вполне очевидно. Например, следующее выражение формируется из двух значений и операции сложения, и оно равно 49:

```
22 + 27
```

Иногда значение не столь очевидно. Например, показанный ниже код — это тоже выражение, потому что формируется из двух значений и операции присваивания:

```
x = 20
```

В C++ значение выражения присваивания определяется как значение его левой части, поэтому данное выражение имеет значение 20. Тот факт, что выражения присваивания имеют значения, означает, что допускаются операторы вроде такого:

```
maids = (cooks = 4) + 3;
```

Выражение `cooks = 4` имеет значение 4, поэтому `maids` присваивается значение 7. Однако то, что C++ допускает подобное поведение, не значит, что вы должны злоупотреблять им. Но то же самое правило, которое разрешает такие специфические операторы, также означает, что разрешен следующий удобный оператор:

```
x = y = z = 0;
```

Это быстрый способ установки одного и того же значения нескольким переменным. Согласно таблице приоритетов (см. приложение Г), присваивание ассоциируется справа налево, поэтому первый `0` присваивается `z`, затем `z = 0` присваивается `y` и т.д.

И, наконец, как упоминалось ранее, выражения отношений, такие как `x < y`, вычисляются как значения `true` и `false` типа `bool`. Короткая программа в листинге 5.3 иллюстрирует некоторые моменты, связанные со значениями выражений. Операция << имеет более высокий приоритет, чем операции, использованные в выражениях, поэтому в коде используются скобки, чтобы задать правильный порядок разбора.

Листинг 5.3. `express.cpp`

```
// express.cpp -- значения выражений
#include <iostream>
int main()
{
    using namespace std;
    int x;
    cout << "The expression x = 100 has the value ";  // вывод значения выражения x = 100
    cout << (x = 100) << endl;
    cout << "Now x = " << x << endl;
    cout << "The expression x < 3 has the value ";  // вывод значения выражения x < 3
    cout << (x < 3) << endl;
    cout << "The expression x > 3 has the value ";  // вывод значения выражения x > 3
    cout << (x > 3) << endl;
    cout.setf(ios_base::boolalpha);                 // новое средство C++
    cout << "The expression x < 3 has the value ";  // вывод значения выражения x < 3
    cout << (x < 3) << endl;
    cout << "The expression x > 3 has the value ";  // вывод значения выражения x > 3
    cout << (x > 3) << endl;
    return 0;
}
```

Ниже показан вывод программы из листинга 5.3:

```
The expression x = 100 has the value 100
Now x = 100
The expression x < 3 has the value 0
The expression x > 3 has the value 1
The expression x < 3 has the value false
The expression x > 3 has the value true
```

Обычно `cout` преобразует значения `bool` в `int` перед тем, как отобразить их, но вызов функции `cout.setf(ios::boolalpha)` устанавливает флаг, который инструктирует `cout` отображать `true` и `false` вместо `1` и `0`.

На заметку!

Выражение C++ — это значение или комбинация значений и операций, и каждое выражение C++ имеет свое значение.

Чтобы вычислить выражение `x = 100`, переменной `x` должно быть присвоено значение `100`. Когда в результате вычисления выражения изменяется значение данных в памяти, мы говорим, что вычисление дает *побочный эффект*. Таким образом, вычисление выражения присваивания в качестве побочного эффекта изменяет значение переменной, которой осуществляется присваивание. Вы можете думать о присваивании как о главном эффекте, но с точки зрения внутреннего устройства C++ первичным эффектом является вычисление выражения. Не все выражения имеют побочные эффекты. Например, вычисление `x + 15` дает новое значение, но не меняет значения `x`. Однако вычисление `++x + 15` дает побочный эффект, т.к. включает в себя инкремент `x`.

От выражения до оператора программы один шаг; для этого достаточно добавить точку с запятой. То есть следующий код будет выражением:

```
age = 100
```

В то же время показанный ниже код является оператором:

```
age = 100;
```

Точнее, это *оператор выражения*. Любое выражение может стать оператором, если к нему добавить точку с запятой, но результат может не иметь смысла с точки зрения программы. Например, если `rodents` — переменная, то следующий код представляет собой допустимый оператор C++:

```
rodents + 6; // допустимое, однако бессмысленное выражение
```

Компилятор разрешает такой оператор, но он не делает ничего полезного. Программа просто вычисляет сумму, ничего с ней не делает и переходит к следующему оператору. (Интеллектуальный компилятор может даже пропустить такой оператор.)

Не выражения и операторы

Некоторые концепции, такие как структуры или цикл `for`, являются ключевыми для понимания C++. Но существуют также относительно менее значимые аспекты синтаксиса, которые иногда могут сбивать с толку, когда уже кажется, что язык вполне понятен. Несколько из них мы сейчас рассмотрим.

Хотя утверждение о том, что добавление точки с запятой к любому выражению превращает его в оператор, справедливо, обратное не верно. То есть исключение точки с запятой из оператора не обязательно преобразует его в выражение. Из всех разновидностей операторов, которые мы рассмотрели до сих пор, операто-

ры возврата, операторы объявления и операторы for не укладываются в правило *оператор = выражение + точка с запятой*. Например, вот оператор:

```
int toad;
```

Однако фрагмент int toad не является выражением и не имеет значения. Это делает следующий код некорректным:

```
eggs = int toad * 1000;    // не верно, это — не выражение
cin >> int toad;           // нельзя комбинировать объявление с cin
```

Подобным же образом нельзя присваивать цикл for переменной. В следующем примере цикл for — это не выражение, поэтому он не имеет значения, и присваивать его не разрешено:

```
int fx = for (i = 0; i< 4; i++)
   cout >> i;              // невозможно
```

Отклонения от правил

Язык C++ добавляет к циклам C возможность, которая требует некоторой поправки к синтаксису цикла for. Исходный синтаксис выглядел следующим образом:

```
for (выражение; выражение; выражение)
  оператор
```

В частности, управляющий раздел конструкции for состоял из трех выражений, разделенных точками с запятой, как это указывалось ранее в настоящей главе. Однако циклы C++ позволяют поступать так, как показано ниже:

```
for (int i = 0; i < 5; i++)
```

То есть в области инициализации цикла for можно объявить переменную. Часто поступать подобным образом очень удобно, но это не вписывается в исходный синтаксис, поскольку объявление не является выражением. Это единственное незаконное поведение, которое подгонялось под правила за счет определения нового вида выражений — *выражения оператора объявления*, которое представляло собой объявление без точки с запятой и могло встречаться только в операторе for. Однако эта поправка была отброшена. Вместо нее решили модифицировать синтаксис оператора for:

```
for (оператор-инициализации-for условие; выражение)
  оператор
```

На первый взгляд это выглядит не совсем понятно, т.к. содержит только одну точку с запятой вместо двух. Но здесь все в порядке, поскольку *оператор-инициализации-for* идентифицируется как оператор, а оператор имеет собственную точку с запятой. Что касается оператора *оператор-инициализации-for*, то он идентифицируется и как выражение-оператор, и как объявление. Это синтаксическое правило заменяет выражение с последующей точкой с запятой оператором, который имеет собственную точку с запятой. Следствием этого является возможность для программистов C++ объявлять и инициализировать переменные внутри оператора цикла for, и они могут теперь выразить все, что нужно, с помощью синтаксиса C++.

С объявлением переменной внутри *оператор-инициализации-for* связан один практический аспект, о котором вы должны знать. Такая переменная существует только внутри оператора for. То есть после того, как программа покидает цикл, переменная исчезает:

```
for (int i = 0; i < 5; i++)
   cout << "C++ knows loop.\n";
cout << i << endl;         // переменная i больше не определена
```

Еще одна вещь, о которой следует знать — это то, что некоторые старые реализации C++ придерживаются старых правил и трактуют приведенный выше цикл, как если бы i была объявлена *перед* циклом, таким образом, делая ее доступной после завершения цикла.

Возврат к циклу for

Давайте попробуем сделать что-то более сложное с помощью цикла. Код в листинге 5.4 использует цикл для вычисления и сохранения первых 16 факториалов. Факториалы, которые являются удобным примером автоматизации обработки, вычисляются следующим образом. Ноль факториал, записываемый как 0!, определен как равный 1. Далее, 1! равен 1*0!, т.е. 1. 2! равно 2*1!, или 2. 3! равно 3*2!, или 6, и т.д. То есть факториал каждого целого числа равен произведению этого числа на факториал предыдущего числа. В программе один цикл используется для вычисления значений последовательных факториалов, с сохранением их в массиве. Второй цикл служит для отображения результатов. Также программа демонстрирует применение внешних объявлений для значений.

Листинг 5.4. formore.cpp

```
// formore.cpp -- дополнительные сведения о циклах for
#include <iostream>
const int ArSize = 16;    // пример внешнего объявления
int main()
{
   long long factorials[ArSize];
   factorials[1] = factorials[0] = 1LL;

   for (int i = 2; i < ArSize; i++)
      factorials[i] = i * factorials[i-1];

   for (i = 0; i < ArSize; i++)
      std::cout << i << "! = " << factorials[i] << std::endl;

   return 0;
}
```

Вывод программы из листинга 5.4 выглядит следующим образом:

```
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
8! = 40320
9! = 362880
10! = 3628800
11! = 39916800
12! = 479001600
13! = 6227020800
14! = 87178291200
15! = 1307674368000
```

Как видите, факториалы растут очень быстро.

На заметку!

В этом листинге используется тип `long long`. Если он не доступен в вашей системе, можете воспользоваться типом `double`. Однако целочисленный формат дает более наглядное визуальное представление о том, насколько быстро растут значения.

Замечания по программе

Программа из листинга 5.4 создает массив для значений факториалов. Элемент 0 хранит 0!, элемент 1 — 1! и т.д. Поскольку первые два факториала равны 1, программа присваивает первым двум элементам массива `factorials` значение `1LL`. (Вспомните, что первый элемент массива имеет индекс 0.) После этого в программе используется цикл для вычисления каждого факториала как произведения индекса на значение предыдущего факториала. Цикл иллюстрирует возможность применения счетчика цикла в его теле как переменной.

Программа из листинга 5.4 демонстрирует, как цикл `for` работает рука об руку с массивами, предоставляя удобное средство доступа к каждому члену массива по очереди. К тому же в `formore.cpp` используется `const` для создания символического представления (`ArSize`) для размера массива. После этого `ArSize` применяется везде, где вступает в игру размер массива — в определении массива, а также в выражении, ограничивающем количество шагов циклов, обрабатывающих массив. Если теперь вы решите расширить программу для вычисления, скажем, 20 факториалов, для этого понадобится только установить `ArSize` в `20` и перекомпилировать программу. Благодаря использованию символической константы, вы избегаете необходимости изменять индивидуально каждое вхождение `16` на `20`.

На заметку!

Определение значения `const` для представления размера массива обычно всегда является хорошей идеей. Это значение `const` можно использовать в объявлении массива и во всех других случаях ссылок на его размер, как, например, в циклах `for`.

Ограничивающее выражение `i < ArSize` отражает тот факт, что индексы элементов массива лежат в пределах от `0` до `ArSize - 1`, т.е. значение индекса должно останавливаться за один шаг до достижения `ArSize`. Вместо него можно было бы использовать проверочное условие `i <= ArSize - 1`, но оно выглядит менее изящно и не меняет сути проверки.

Обратите внимание, что в программе объявляется переменная `ArSize` типа `const int` вне тела функции `main()`. Как упоминалось в конце главы 4, это делает `ArSize` внешними данными. Объявление `ArSize` в подобной манере имеет два последствия: `ArSize` существует на протяжении всего времени жизни программы, и все функции в файле программы могут использовать `ArSize`. В данном конкретном случае в программе присутствует только одна функция, поэтому внешнее объявление `ArSize` не имеет особого практического смысла. Однако программы с множеством функций часто выигрывают от совместного доступа к внешним константам, поэтому дальше мы еще попрактикуемся в их применении.

Кроме того, в этом примере напоминается о том, что для доступа к выбранным стандартным именам можно использовать `std::` вместо директивы `using`.

Изменение шага цикла

До сих пор в примерах циклов счетчик цикла увеличивался или уменьшался на единицу на каждом шаге. Это можно изменить, модифицировав обновляющее выражение. Программа в листинге 5.5, например, увеличивает счетчик цикла на величину

введенного пользователем шага. Вместо применения i++ в качестве обновляющего выражения она использует выражение i = i + by, где by — выбранный пользователем шаг цикла.

Листинг 5.5. bigstep.cpp

```
// bigstep.cpp -- цикл указанным пользователем шагом
#include <iostream>
int main()
{
   using std::cout;                // объявление using
   using std::cin;
   using std::endl;
   cout << "Enter an integer: ";   // ввод целого числа
   int by;
   cin >> by;
   cout << "Counting by " << by << "s:\n";
   for (int i = 0; i < 100; i = i + by)
      cout << i << endl;
   return 0;
}
```

Ниже показан пример запуска программы из листинга 5.5:

```
Enter an integer: 17
Counting by 17s:
0
17
34
51
68
85
```

Когда i достигает значения 102, цикл завершается. Главное, на что здесь нужно обратить внимание: в качестве обновляющего выражения можно использовать любое допустимое выражение. Например, если вы захотите на каждом шаге цикла возводить i в квадрат и прибавлять 10, можете воспользоваться выражением i = i * i + 10.

Другой момент, который следует отметить: часто лучше проверять на предмет неравенства, чем равенства. Например, проверка i == 100 в этом примере не подойдет, поскольку i перепрыгивает через значение 100.

Наконец, в этом примере иллюстрируется применение объявлений using вместо директивы using.

Доступ внутрь строк с помощью цикла for

Цикл for предоставляет прямой способ доступа к каждому символу в строке. Например, программа в листинге 5.6 позволяет ввести строку и отобразить ее символ за символом в обратном порядке. В этом примере можно использовать либо объект класса string, либо массив char, потому что оба позволяют применять нотацию массивов для доступа к индивидуальным символам строки. В листинге 5.6 применяется объект класса string. Метод size() класса string возвращает количество символов в строке; цикл использует это значение в выражении инициализации для установки i в индекс последнего символа строки, исключая нулевой символ. Для выполнения обратного отсчета в программе применяется операция декремента (--), уменьшающая значение индекса массива на каждом шаге цикла. Также в листинге 5.6 используется

операцию сравнения "больше или равно" (>=), чтобы проверить, достиг ли цикл первого элемента. Чуть позже мы подведем итог по всем операциям сравнения.

Листинг 5.6. forstr1.cpp

```
// forstr1.cpp -- использование цикла for для строки
#include <iostream>
#include <string>
int main()
{
    using namespace std;
    cout << "Enter a word: ";
    string word;
    cin >> word;
    // Отображение символов в обратном порядке
    for (int i = word.size() - 1; i >= 0; i--)
    cout << word[i];
    cout << "\nBye.\n";
    return 0;
}
```

Ниже показан пример запуска программы из листинга 5.6:

```
Enter a word: animal
lamina
Bye.
```

Как видите, программа действительно успешно напечатала слово `animal` в обратном порядке; выбор этого слова в качестве теста яснее демонстрирует эффект от работы программы, нежели выбор палиндрома наподобие `rotator`, `redder` или `stats`.

Операции инкремента и декремента

Язык C++ снабжен несколькими операциями, которые часто используются в циклах; давайте потратим немного времени на их изучение. Вы уже видели две из них: операция инкремента (++), которая получила отражение в самом названии C++, а также операция декремента (--). Эти операции выполняют два чрезвычайно часто встречающихся действия в циклах: увеличивают и уменьшают на единицу значение счетчика цикла. Однако к тому, что вы уже знаете о них, есть что добавить. Каждая из них имеет два варианта. *Префиксная* версия операции указывается перед операндом, как в `++x`. *Постфиксная* версия следует после операнда, как в `x++`. Эти две версии имеют один и тот же эффект для операнда, но отличаются в контексте применения. Все похоже на получение оплаты за стрижку газона авансом или после завершения работы: оба метода имеют один и тот же конечный результат для вашего бумажника, но отличаются тем, в какой момент деньги в него добавляются. В листинге 5.7 демонстрируется разница на примере операции инкремента.

Листинг 5.7. plus_one.cpp

```
// plus_one.cpp -- операция инкремента
#include <iostream>
int main()
{
    using std::cout;
    int a = 20;
    int b = 20;
```

```
    cout << "a   = " << a << ":   b = " << b << "\n";
    cout << "a++ = " << a++ << ": ++b = " << ++b << "\n";
    cout << "a   = " << a << ":   b = " << b << "\n";
    return 0;
}
```

Результат выполнения этой программы показан ниже:

```
a   = 20:   b = 20
a++ = 20: ++b = 21
a   = 21:   b = 21
```

Грубо говоря, нотация `a++` означает "использовать текущее значение `a` при вычислении выражения, затем увеличить `a` на единицу". Аналогично, нотация `++a` означает "сначала увеличить значение `a` на единицу, затем использовать новое значение при вычислении выражения". Например, мы имеем следующие отношения:

```
int x = 5;
int y = ++x;              // изменить x, затем присвоить его y
                          // y равно 6, x равно 6
int z = 5;
int y = z++;              // присвоить y, затем изменить z
                          // y равно 5, z равно 6
```

Операции инкремента и декремента представляют собой простой удобный способ решения часто возникающей задачи увеличения или уменьшения значений на единицу.

Операции инкремента и декремента — симпатичные и компактные, но не стоит поддаваться соблазну и применять их к одному и тому же значению более одного раза в одном и том же операторе. Проблема в том, что при этом правила "использовать и изменить" и "изменить и использовать" становятся неоднозначными. То есть, следующий оператор в различных системах может дать совершенно разные результаты:

```
x = 2 * x++ * (3 - ++x);   // не поступайте так
```

В C++ поведение операторов подобного рода не определено.

Побочные эффекты и точки следования

Давайте посмотрим внимательнее на то, что в C++ говорится, и что не говорится о том, когда операции инкремента вступают в силу. Для начала вспомните, что *побочный эффект* — это эффект, который проявляется, когда вычисление выражения приводит к модификации чего-либо, например, значения переменной. *Точка следования* (sequence point) — это точка при выполнении программы, где все побочные эффекты гарантированно будут завершены, прежде чем программа перейдет к следующему шагу. В C++ точка с запятой в операторе отмечает точку следования. Это значит, что все изменения, выполненные операциями присваивания, инкремента и декремента в операторе, должны произойти, прежде чем программа перейдет к следующему оператору. Некоторые операторы, рассматриваемые в последующих разделах, имеют точки следования. К тому же конец любого полного выражения представляет точку следования.

Что такое полное выражение? Это выражение, которое не является частью более крупного выражения. Примеры полных выражений включают часть выражения в операторе выражения (без точки с запятой), а также выражение, служащее проверочным условием в цикле `while`.

Точки следования помогают прояснить, когда выполняется постфиксный инкремент. Рассмотрим, например, следующий код:

```
while (guests++ < 10)
   cout << guests << endl;
```

(Цикл `while`, рассматриваемый позже в этой главе, работает подобно циклу, в котором имеется только проверочное выражение.) Иногда новички в C++ предполагают, что "использовать значение, затем увеличить его" означает в данном контексте увеличение `guests` после того, как эта переменная использована в операторе `cout`. Однако `guests++ < 10` является полным выражением, поскольку это проверочное условие цикла `while`, поэтому конец этого выражения представляет собой точку следования. Таким образом, C++ гарантирует, что побочный эффект (инкрементирование `quests`) произойдет перед тем, как программа перейдет к `cout`. Применение постфиксной формы гарантирует, что `quests` увеличится после сравнения с `10`.

Теперь рассмотрим следующий оператор:

```
y = (4 + x++) + (6 + x++);
```

Выражение `4 + x++` не является полным выражением, поэтому C++ не гарантирует, что значение `x` будет увеличено немедленно после вычисления вложенного выражения `4 + x++`. Здесь полным выражением является весь оператор присваивания, и точка с запятой отмечает точку следования, поэтому все, что гарантирует C++ — это то, что `x` будет увеличено дважды перед тем, как программа перейдет к следующему оператору. C++ не указывает, будет ли переменная `x` инкрементирована после вычисления каждого подвыражения либо после вычисления всего выражения в целом. Поэтому вы должны избегать операторов такого рода.

В документации по C++11 понятие точки следования было отброшено, поскольку эта концепция не особенно хорошо вписывается в обсуждение множества потоков выполнения. Вместо этого описания помещены в рамки терминов последовательной обработки, когда ряд событий рассматриваются как последовательность перед другими событиями. Такой описательный подход не предназначен для изменения правил; его цель заключается в предоставлении языка, который может более четко обрабатывать многопоточное программирование.

Сравнение префиксной и постфиксной форм

Понятно, что разница между префиксной и постфиксной формами операций проявляется, если значение их операнда используется для каких-то целей — в качестве аргумента функции или для присваивания переменной. Но что если инкрементируемое или декрементируемое значение не используется? Например, отличаются ли

```
x++;
```

и

```
++x;
```

друг от друга? Или же отличаются друг от друга

```
for (n = lim; n > 0; --n)
   ...;
```

и

```
for (n = lim; n > 0; n--)
   ...;
```

Рассуждая логически, можно предположить, что в этих двух ситуациях префиксная форма не отличается от постфиксной. Значения выражений не используются, поэтому единственный эффект от них — побочный, т.е. увеличение или уменьшение операнда. Здесь выражения, использующие эти операции, являются полными выражениями, поэтому побочный эффект от инкремента x и декремента n гарантированно будет получен на момент, когда программа переходит к следующему шагу; префиксная и постфиксная формы дают один и тот же результат.

Однако, несмотря на то, что выбор между двумя формами никак не отражается на поведении программы, все же он может несколько повлиять на скорость выполнения. Для встроенных типов и современных компиляторов это может показаться неважным. Но C++ разрешает определять самостоятельно эти операции для классов. В таком случае пользователь определяет префиксную функцию, которая работает, увеличивая значение и затем возвращая его. Постфиксная версия работает, сначала запоминая копию значения, увеличивает его и возвращает сохраненную копию. Таким образом, для классов префиксная версия немного более эффективна, чем постфиксная.

Короче говоря, для встроенных типов, скорее всего, между двумя формами этих операций разницы нет. Но для типов, определенных пользователем, оснащенных операциями инкремента и декремента, префиксная форма более эффективна.

Операции инкремента и декремента и указатели

Вы можете использовать операции инкремента с указателями так же, как и с базовыми переменными. Вспомните, что добавление единицы к указателю увеличиваст его на количество байт, представляющих размер указываемого типа. То же правило остается в силе при инкременте и декременте указателей:

```
double arr[5] = {21.1, 32.8, 23.4, 45.2, 37.4};
double *pt = arr;     // pt указывает на arr[0], т.е. на 21.1
++pt;                 // pt указывает на arr[1], т.е. на 32.8
```

Эти операции также можно применять для изменения значений, на которые указывают указатели, используя их в сочетании с операцией *. Применение * и ++ к указателю вызывает вопросы о том, что собственно должно разыменовываться, а что — инкрементироваться. Префиксный инкремент, префиксный декремент и операция разыменования имеют одинаковый приоритет и ассоциируются слева направо. Постфиксный инкремент и декремент имеют одинаковый приоритет, более высокий, чем приоритет префиксных форм. Эти две операции также ассоциируются слева направо.

Правило ассоциации слева направо для префиксных операций подразумевает, что в записи *++pt сначала применяется операция ++ к pt (потому что ++ находится справа от *), а затем к новому значению pt применяется операция *:

```
double x = *++pt;     // инкремент указателя, получение значения;
                      // т.е. arr[2], или 23.4
```

С другой стороны, ++*pt означает — получить значение, на которое указываст pt, а затем увеличить указатель:

```
++*pt;      // инкремент указываемого значения; т.е. изменение 23.4 на 24.4
```

Здесь pt по-прежнему будет указывать на arr[2].

Теперь рассмотрим следующую комбинацию:

```
(*pt)++;    // инкремент указываемого значения
```

Скобки отражают то, что сначала выполняется разыменование указателя, выдавая значение 24.4. Затем операция ++ увеличивает значение до 25.4; pt по-прежнему указывает на arr[2]. И, наконец, рассмотрим такую комбинацию:

```
x = *pt++;  // разыменование исходного указателя, затем инкремент указателя
```

Более высокий приоритет постфиксной операции ++ означает, что ++ увеличит pt, а не *pt, поэтому инкремент касается указателя. Но тот факт, что использована постфиксная операция, означает, что разыменовываться будет исходный адрес, т.е. &arr[2], а не новый адрес. Таким образом, значение *pt++ равно arr[2], или 25.4, но после завершения оператора pt будет указывать на arr[3].

На заметку!

Инкрементирование и декрементирование указателей следует правилам арифметики указателей. То есть, если pt указывает на первый элемент массива, ++pt изменяет его так, что он после этого указывает на второй его элемент.

Комбинация операций присваивания

В листинге 5.5 используется следующее выражение для обновления счетчика цикла:

```
i = i + by
```

В C++ предусмотрена комбинированная операция сложения с присваиванием, которая позволяет получить тот же результат, но более кратко:

```
i += by
```

Операция += складывает значение своих двух операндов и присваивает результат левому операнду. Это предполагает, что левый операнд должен быть чем-то таким, чему можно присваивать значения, вроде переменной, элемента массива, члена структуры либо элемента данных, полученного через разыменование указателя:

```
int k = 5;
k += 3;                    // нормально, k установлено в 8
int *pa = new int[10];     // pa указывает на pa[0]
pa[4] = 12;
pa[4] += 6;                // нормально, pa[4] установлено в 18
*(pa + 4) += 7;            // нормально, pa[4] установлено в 25
pa += 2;                   // нормально, pa указывает на бывший pa[2]
34 += 10;                  // ошибка!
```

Каждая арифметическая операция имеет соответствующую операцию присваивания, как показано в табл. 5.1. Каждая такая операция работает аналогично +=. То есть, например, следующий оператор заменяет текущее значение k в 10 раз большим значением:

```
k *= 10;
```

Таблица 5.1. Комбинированные операции присваивания

Операция	Эффект (L — левый операнд, R — правый операнд)
+=	Присваивает L + R операнду L
-=	Присваивает L - R операнду L
*=	Присваивает L * R операнду L
/=	Присваивает L / R операнду L
%=	Присваивает L % R операнду L

Составные операторы, или блоки

Формат — или синтаксис — написания оператора `for` может показаться чересчур ограниченным, поскольку тело цикла должно состоять из всего лишь одного оператора. Это весьма неудобно, если вы хотите, чтобы тело цикла включало несколько операторов. К счастью, C++ предлагает синтаксис, позволяющий поместить в тело цикла любое количество операторов. Трюк заключается в использовании пары фигурных скобок, с помощью которых конструируется *составной оператор*, или *блок*. Блок состоит из пары фигурных скобок с заключенными между ними операторами и синтаксически воспринимается как один оператор. Например, в листинге 5.8 фигурные скобки применяются для комбинирования трех отдельных операторов в единый блок. Это позволяет в теле цикла организовать вывод приглашения пользователю, прочитать его ввод и выполнить вычисления. Программа вычисляет текущую сумму вводимых значений, и это является удобным случаем для использования операции +=.

Листинг 5.8. `block.cpp`

```
// block.cpp -- использование блока
#include <iostream>
int main()
{
   cout << "The Amazing Accounto will sum and average ";
   cout << "five numbers for you.\n";
   cout << "Please enter five values:\n";
   double number;
   double sum = 0.0;
   for (int i = 1; i <= 5; i++)
   {                                                  // начало блока
      cout << "Value " << i << ": ";                  // ввод числа
      cin >> number;
      sum += number;
   }                                                  // конец блока
   cout << "Five exquisite choices indeed! ";
   cout << "They sum to " << sum << endl;             // вывод суммы
   cout << "and average to " << sum / 5 << ".\n";     // вывод среднего значения
   cout << "The Amazing Accounto bids you adieu!\n";
   return 0;
}
```

Ниже показан пример запуска программы из листинга 5.8:

```
The Amazing Accounto will sum and average five numbers for you.
Please enter five values:
Value 1: 1942
Value 2: 1948
Value 3: 1957
Value 4: 1974
Value 5: 1980
Five exquisite choices indeed! They sum to 9801
and average to 1960.2.
The Amazing Accounto bids you adieu!
```

Предположим, что вы сохранили отступы, но удалили фигурные скобки:

```
for (int i = 1; i <= 5; i++)
   cout << "Value " << i << ": ";
   cin >> number;
   sum += number;
cout << "Five exquisite choices indeed! ";
```

Компилятор игнорирует отступы, поэтому в тело цикла попадет только первый оператор. То есть цикл напечатает пять приглашений и ничего более. После завершения цикла программа перейдет к выполнению последующих строк, читая и суммируя только одно значение.

Составные операторы обладают еще одним интересным свойством. Если вы определяете новую переменную внутри блока, она будет существовать только во время выполнения операторов этого блока. Когда поток выполнения покидает блок, такая переменная уничтожается. Это значит, что переменная известна только внутри блока.

```
#include <iostream>
int main()
{
   int x = 20;
   {                                  // начало блока
      int y = 100;
      cout << x << endl;              // нормально
      cout << y << endl;              // нормально
   }                                  // конец блока
   cout << x << endl;                 // нормально
   cout << y << endl;                 // ошибка во время компиляции!
   return 0;
}
```

Обратите внимание, что переменная, объявленная вне блока, определена и внутри него.

А что случится, если вы объявите внутри блока переменную с тем же именем, что и у одной из объявленных вне блока? Новая переменная скроет старую, начиная с точки ее появления и до конца блока. Затем старая переменная опять станет видимой, как в следующем примере:

```
#include <iostream>
int main()
{
   using std::cout;
   using std::endl;
   int x = 20;                        // исходная переменная x
   {                                  // начало блока
      cout << x << endl;              // используется исходная переменная x
      int x = 100;                    // новая переменная x
      cout << x << endl;              // используется новая переменная x
   }                                  // конец блока
   cout << x << endl;                 // используется исходная переменная x
   return 0;
}
```

Дополнительные синтаксические трюки — операция запятой

Как вы уже видели, блок позволяет помещать два и более оператора там, где синтаксис C++ разрешает лишь один. Операция запятой (`,`) делает то же самое с выражениями, позволяя вставлять два выражения туда, где синтаксис C++ допускает только одно. Например, предположим, что имеется цикл, в котором на каждом шаге одна переменная увеличивается на единицу, а вторая — на единицу уменьшается. Было бы удобно сделать то и другое в обновляющей части цикла `for`, но синтаксис цикла разрешает там только одно выражение. Решение состоит в применении операции запятой для комбинации двух выражений в одно:

```
++j, --i     // два выражения воспринимаются как одно
             // для удовлетворения требований синтаксиса
```

Запятая — это не всегда операция. Например, запятая в следующем объявлении служит для разделения имен в списке объявляемых переменных:

```
int i, j;   // здесь запятая — разделитель, а не операция
```

В листинге 5.9 операция запятой используется дважды в программе, которая переставляет содержимое объекта класса `string`. (Написать программу можно также с использованием массива `char`, но длина слова должна ограничиваться определенным размером массива.) Обратите внимание, что код в листинге 5.6 отображает содержимое массива в обратном порядке, а код в листинге 5.9 на самом деле перемещает символы по кругу в пределах массива. Программа из листинга 5.9 также использует блок для объединения нескольких операторов в один.

Листинг 5.9. `forstr2.cpp`

```
// forstr2.cpp -- обращение порядка элементов массива
#include <iostream>
#include <string>
int main()
{
   using namespace std;
   cout << "Enter a word: ";
   string word;
   cin >> word;

   // Физическая модификация объекта string
   char temp;
   int i, j;
   for (j = 0, i = word.size() - 1; j < i; --i, ++j)
   {                            // начало блока
      temp = word[i];
      word[i] = word[j];
      word[j] = temp;
   }                            // конец блока
   cout << word << "\nDone\n";
   return 0;
}
```

Ниже приведен пример выполнения программы из листинга 5.9:

```
Enter a word: stressed
desserts
Done
```

Кстати, класс `string` предлагает более удобный способ обращения строки, но мы отложим его описание до главы 16.

Замечания по программе

Взгляните еще раз на управляющий раздел цикла `for` в программе 5.9. Во-первых, в нем используется операция запятой для указания последовательно двух инициализаций в одном выражении — первой части управляющего раздела. Во-вторых, в нем операция запятой применяется еще раз для комбинирования двух обновлений в единственное выражение для последней части управляющего раздела.

Далее посмотрите на тело цикла. В программе используются фигурные скобки для комбинации нескольких операторов в одно целое. В теле программы выполняется изменение порядка символов в слове на противоположный, для чего меняются местами первый и последний элементы массива. Затем инкрементируется `j` и декрементируется `i`, так что теперь они ссылаются на элемент, следующий за первым, и элемент, предшествующий последнему, соответственно. После этого указанные элементы меняются местами. Обратите внимание, что проверочное условие `j < i` обеспечивает останов цикла, когда он достигает центра массива. Если бы он продолжался дальше, то начался бы обратный обмен символов на их старые места (рис. 5.2).

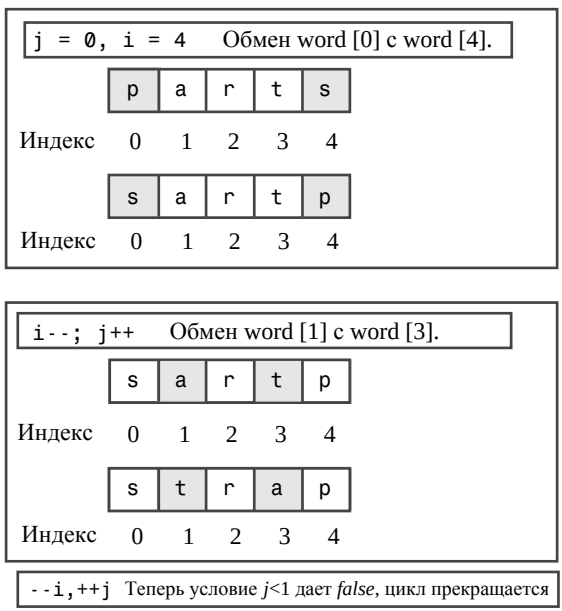


Рис. 5.2. *Обращение строки*

Здесь следует отметить еще одну вещь — местоположение объявлений переменных `temp`, `i` и `j`. В коде переменные `i` и `j` объявлены перед началом цикла, поскольку комбинировать два объявления подряд, разделяя их операцией запятой, нельзя. Причина в том, что объявления уже используют запятую для другой цели — в качестве разделителя элементов списка. Можно использовать один оператор-объявление для создания и инициализации двух переменных, но это выглядит несколько запутано:

```
int j = 0, i = word.size() - 1;
```

В этом случае запятая служит просто разделителем в списке, а не операцией, поэтому выражение объявляет и инициализирует обе переменных. Тем не менее, смотрится это так, будто объявлена только `j`.

Между прочим, вы можете объявить `temp` внутри цикла `for`:

```
int temp = word[i];
```

Это может привести к тому, что `temp` будет размещаться и освобождаться на каждом шаге цикла. В результате работа программы может замедлиться по сравнению с вариантом, когда переменная `temp` объявлена один раз, перед началом цикла. С другой стороны, при объявлении внутри цикла после его завершения `temp` уничтожается.

Особенности операции запятой

До сих пор операция запятой чаще всего использовалась для размещения двух или более выражений в одном выражении цикла `for`. Но C++ снабжает эту операцию двумя дополнительными свойствами. Во-первых, операция гарантирует, что первое выражение вычисляется перед вторым. (Другими словами, операция запятой — это точка следования.) Выражения наподобие следующего вполне безопасны:

```
i = 20, j = 2 * i     // i присваивается 20, затем j получает значение 40
```

Во-вторых, C++ устанавливает, что значением выражения с запятой является значение его второй части. Скажем, значение предыдущего выражения равно `40`, потому что таково значение выражения `j = 2 * i`.

Операция запятой обладает наименьшим приоритетом среди всех операций. Например, следующий оператор:

```
cats = 17,240;
```

читается как

```
(cats = 17), 240;
```

То есть `cats` присваивается `17`, а `240` не делает ничего. Но поскольку скобки имеют более высокий приоритет, то приведенный ниже оператор дает результат `240` — выражение, находящееся справа от запятой:

```
cats = (17,240);
```

Выражения отношений

Компьютеры — это нечто большее, чем неутомимые обработчики чисел. Они также умеют сравнивать значения, и эта их способность лежит в основе автоматического принятия решений. В C++ эту возможность реализуют операции отношений. В C++ доступны шесть операций отношений для сравнения чисел. Поскольку символы представлены своими ASCII-кодами, эти операции можно также применять и для сравнения символов. Они не работают со строками в стиле C, но работают с объектами класса `string`. Каждое сравнивающее выражение возвращает булевское (типа `bool`) значение `true`, если сравнение истинно, и `false` — в противном случае, поэтому данные операции хорошо подходят для применения в проверочных условиях циклов. (Старые реализации оценивали истинные выражения как `1` и ложные — как `0`.)

В табл. 5.2 предлагается список операций отношений.

Таблица 5.2. Операции отношений

Операция	Описание
<	Меньше чем
<=	Меньше или равно
==	Равно
>	Больше чем
>=	Больше или равно
!=	Не равно

Этими шестью операциями отношений исчерпываются все возможности, предусмотренные в C++ для сравнения чисел. Если хотите сравнить два значения на предмет того, какое из них более красивое или более удачное, вам придется поискать в другом месте.

Вот некоторые примеры сравнений:

```
for (x = 20; x > 5; x--)            // продолжать, пока x больше чем 5
for (x = 1; y != x; ++x)            // продолжать, пока y не равно x
for (cin >> x; x == 0; cin >> x))   // продолжать, пока x равно 0
```

Операции отношений обладают более низким приоритетом, нежели арифметические операции. Это значит, что следующее выражение:

```
x + 3 > y - 2          // выражение 1
```

соответствует такому:

```
(x + 3) > (y - 2)      // выражение 2
```

и не соответствует этому:

```
x + (3 > y) - 2        // выражение 3
```

Поскольку выражение `(3 > y)` после приведения значения `bool` к типу `int` даст либо `0`, либо `1`, выражения 2 и 3 корректны. Но большинство из нас подразумевают под выражением 1 то, что записано в выражении 2; так поступает и C++.

Присваивание, сравнение и вероятные ошибки

Не следует путать операцию проверки равенства (==) с операцией присваивания (=). Следующее выражение задает вопрос "Равно ли значение `musicians` четырем?":

```
musicians == 4         // сравнение
```

Выражение может принимать значение `true` или `false`. Приведенное ниже выражение присваивает `musicians` значение `4`:

```
musicians = 4          // присваивание
```

Полное выражение в данном случае имеет значение `4`, потому что таково значение левой части.

Гибкий синтаксис цикла `for` создает любопытную возможность ошибки. Если вы непреднамеренно удалите один символ = из операции сравнения == и примените операцию присваивания вместо операции сравнения в проверочной части цикла `for`, это будет расценено компилятором как вполне правильный код. Причина в том, что в качестве проверочного условия цикла `for` можно использовать любое допустимое выражение C++. Вспомните, что ненулевые значения оцениваются как `true`, а нулевые — как `false`. Выражение, которое присваивает `4` переменной `musicians`, имеет значение `4` и трактуется как `true`. Если вы перешли в C++ от таких языков, как Pascal или BASIC, в которых для проверки равенства используется операция =, то вы будете склонны к ошибкам подобного рода.

В листинге 5.10 показана ситуация, когда есть риск допустить такую ошибку. Программа пытается проверить массив `quizscores` и останавливается, когда достигает первого значения, которое не равно `20`. Сначала демонстрируется цикл, который корректно использует сравнение, а затем — цикл, в котором в проверочном условии ошибочно вместо операции равенства использована операция присваивания. Помимо этой, программа также содержит в себе еще одну вопиющую ошибку, исправление которой будет показано позже. (На ошибках учатся, и листинг 5.10 поможет в этом.)

Листинг 5.10. equal.cpp

```
// equal.cpp -- равенство или присваивание
#include <iostream>
int main()
{
   using namespace std;
   int quizscores[10] =
      { 20, 20, 20, 20, 20, 19, 20, 18, 20, 20};
   cout << "Doing it right:\n";              // правильно
   int i;
   for (i = 0; quizscores[i] == 20; i++)
      cout << "quiz " << i << " is a 20\n";
   // Предупреждение: возможно, лучше почитать об этой программе,
   // чем в действительности запускать ее.
   cout << "Doing it dangerously wrong:\n";  // неправильно
   for (i = 0; quizscores[i] = 20; i++)
      cout << "quiz " << i << " is a 20\n";
   return 0;
}
```

Поскольку с программой в листинге 5.10 связана серьезная проблема, возможно, лучше почитать о ней, а не запускать. Вот как будет выглядеть вывод:

```
Doing it right:
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
Doing it dangerously wrong:
quiz 0 is a 20
quiz 1 is a 20
quiz 2 is a 20
quiz 3 is a 20
quiz 4 is a 20
quiz 5 is a 20
quiz 6 is a 20
quiz 7 is a 20
quiz 8 is a 20
quiz 9 is a 20
quiz 10 is a 20
quiz 11 is a 20
quiz 12 is a 20
quiz 13 is a 20
...
```

Первый цикл корректно завершается после отображения первых пяти значений из массива. Но второй цикл начинает с отображения всего массива. Хуже того, он сообщает, что все значения в нем равны 20. И еще хуже: он не останавливается по достижении конца массива! Ну, и самое неприятное то, что программа может (хотя и не обязательно) препятствовать выполнению других приложений, которые функционировали в системе, и потребовать перезагрузки компьютера.

Что здесь неправильно — это, конечно же, содержимое проверочного условия:

```
quizscores[i] = 20
```

Во-первых, поскольку здесь элементу массива присваивается ненулевое значение, выражение всегда вычисляется как истинное. Во-вторых, т.к. это выражение присваи-

вает значения элементам массива, оно на самом деле изменяет данные. В-третьих, из-за того, что проверочное условие остается истинным, программа продолжает изменять данные и за концом массива. Оно просто продолжает и продолжает вставлять в память значения 20! Это нехорошо.

Проблема с ошибками подобного рода состоит в том, что код синтаксически корректен, поэтому компилятор не может распознать ошибку. (Однако годы и годы *программирования на C и C++ заставили многих поставщиков компиляторов, по крайней* мере, выдавать предупреждения, которые запрашивают, действительно ли вы имеете в виду то, что написали.)

Внимание!

Не применяйте = для проверки равенства, используйте ==.

Подобно C, язык C++ предлагает бо́льшую свободу, чем многие другие языки программирования. Это даром не обходится — на разработчика возлагается более высокая ответственность. Ничто другое, кроме тщательного планирования, не предохранит вашу программу от выхода за границы стандартного массива C++. Однако, используя классы C++, вы можете проектировать безопасные типы массивов, которые предохранят от подобной бессмыслицы. Пример этого приведен в главе 13. А пока вы должны встраивать защиту в свои программы, когда нуждаетесь в ней. Например, цикл в листинге 5.10 должен включать проверку, которая не позволит ему выйти за пределы массива. Это верно даже для "хороших" циклов. Если все элементы массива из этого примера содержали бы значение 20, то этот самый "хороший" цикл также вышел бы за границы массива. Короче говоря, цикл должен проверять как значения массива, так и индекс. В главе 6 будет показано, как использовать логические операции для комбинирования таких проверок в единое условие.

Сравнение строк в стиле C

Предположим, что вы хотите узнать, хранится ли в символьном массиве слово `mate`. Если `word` — имя массива, то следующая проверка не сделает того, что вы ожидаете:

```
word == "mate"
```

Вспомните, что имя массива — это синоним его адреса. Аналогично, строковая константа в двойных кавычках является синонимом ее адреса. Таким образом, приведенное выражение сравнения не проверяет идентичность строк; оно проверяет, находятся ли они по одному и тому же адресу. Ответом будет — нет, даже если эти две строки состоят из одинаковых символов.

Поскольку C++ обрабатывает строки в стиле C как адреса, вы мало что получите, если попытаетесь воспользоваться операциями отношений для сравнения строк. Вместо этого можете обратиться к библиотеке строк в стиле C и применять для их сравнения функцию `strcmp()`. Эта функция принимает в виде аргументов два адреса строк. Это значит, что аргументы могут быть указателями, строковыми константами либо именами символьных массивов. Если две строки идентичны, функция возвращает значение `0`. Если первая строка предшествует второй в алфавитном порядке, `strcmp()` возвращает отрицательное значение, если же первая строка следует за второй в алфавитном порядке, то `strcmp()` возвращает положительное значение. Говорить "в последовательности сопоставления в системе" будет более точно, нежели "в алфавитном порядке". Это значит, что символы сравниваются в соответствии с их системными кодами. Например, в коде ASCII заглавные буквы имеют меньшие коды, чем строчные, поэтому заглавные буквы предшествуют строчным в порядке сорти-

ровки. То есть строка "Zoo" предшествует строке "aviary". Тот факт, что сравнение основано на значениях кодов, также означает, что заглавные и строчные буквы отличаются, поэтому строка "FOO" отличается от "foo".

В некоторых языках, таких как BASIC и стандартный Pascal, строки, сохраненные в массивах разных размеров, по определению не равны друг другу. Но строки в стиле C ограничиваются нулевым символом, а не размером содержащего их массива. Это значит, что две строки могут быть идентичными, даже если содержатся в массивах разного размера:

```
char big[80] = "Daffy";         // 5 букв плюс \0
char little[6] = "Daffy";       // 5 букв плюс \0
```

Кстати, хотя и нельзя применять операции отношений для сравнения строк, вы можете использовать их для сравнения символов, потому что символы относятся к целочисленным типам. Поэтому следующий код является допустимым, по крайней мере, для наборов символов ASCII и Unicode, для отображения символов по алфавиту:

```
for (ch = 'a'; ch <= 'z'; ch++)
   cout << ch;
```

Программа в листинге 5.11 использует `strcmp()` в проверочном условии цикла `for`. Эта программа отображает слово, изменяет его первую букву, отображает его снова и продолжает это делать до тех пор, пока `strcmp()` не определит, что `word` содержит строку "matė". Обратите внимание на включение файла `cstring` — в нем содержится прототип `strcmp()`.

Листинг 5.11. compstr1.cpp

```
// compstr1.cpp -- сравнение строк с использованием массивов
#include <iostream>
#include <cstring>  // прототип для strcmp()
int main()
{
  using namespace std;
  char word[5] = "?ate";
  for (char ch = 'a'; strcmp(word, "mate"); ch++)
  {
    cout << word << endl;
    word[0] = ch;
  }
  cout << "After loop ends, word is " << word << endl; // вывод word по завершении цикла
  return 0;
}
```

Ниже показан вывод программы из листинга 5.11:

```
?ate
aate
bate
cate
date
eate
fate
gate
hate
iate
jate
kate
late
After loop ends, word is mate
```

Замечания по программе

Программа в листинге 5.11 содержит несколько интересных моментов. Один из них, конечно же — проверка цикла. Вы хотите, чтобы цикл продолжался до тех пор, пока `word` не совпадает с `"mate"`. То есть до тех пор, пока `strcmp()` сообщает, что строки не одинаковы. Наиболее очевидный способ сделать это выглядит следующим образом:

```
strcmp(word, "mate") != 0 // строки не одинаковы
```

Этот оператор имеет значение `1` (`true`), если строки не одинаковы, и значение `0` (`false`), если они совпадают. Но как насчет самого вызова `strcmp(word, "mate")`? Он возвращает ненулевое значение (`true`), если строки отличаются, и `0` (`false`) — если строки эквивалентны. По сути, функция возвращает `true`, если строки разные, и `false`, если они одинаковы. Вы можете использовать только саму функцию вместо всего сравнивающего выражения. Это даст тот же результат при меньшем объеме кода. К тому же, это — традиционный способ применения `strcmp()` в языках C и C++.

Проверка на эквивалентность или порядок

Функцию `strcmp()` можно применять для проверки строк в стиле C на эквивалентность или порядок. Следующее выражение истинно, если `str1` и `str2` идентичны:

```
strcmp(str1, str2) == 0
```

Выражения

```
strcmp(str1, str2) != 0
```

и

```
strcmp(str1, str2)
```

истинны, когда `str1` и `str2` не идентичны. Показанное ниже выражение истинно, если `str1` по порядку предшествует `str2`:

```
strcmp(str1, str2) < 0
```

И, наконец, следующее выражение истинно, когда `str1` следует за `str2`:

```
strcmp(str1, str2) > 0
```

Таким образом, функция `strcmp()` может играть роль операций `==`, `!=`, `<` и `>`, в зависимости от того, как будет составлено проверочное условие.

Далее в `compstr1.cpp` используется операция инкремента для прохода переменной `ch` по всему алфавиту:

```
ch++
```

Операции инкремента и декремента можно применять в отношении символьных переменных, т.к. тип `char` на самом деле является целочисленным, поэтому данная операция в действительности изменяет целочисленный код, хранящийся в переменной. К тому же обратите внимание, что использование индекса массива упрощает изменение отдельных символов в строке:

```
word[0] = ch;
```

Сравнение строк класса `string`

Жизнь станет немного легче, если вместо строк в стиле C использовать строки класса `string`, поскольку этот класс позволяет применять операции отношений для выполнения сравнений. Это становится возможным благодаря определению функций класса, которые "перегружают", или переопределяют, операции. В главе 12 будет

показано, как включать это средство в классы, но с практической точки зрения все, что вам нужно знать сейчас — это то, что с объектами класса `string` можно использовать операции сравнения. В листинге 5.12 представлен преобразованный код из листинга 5.11, в котором вместо массива `char` применяется объект `string`.

Листинг 5.12. `compstr2.cpp`

```
// compstr2.cpp -- сравнение строк с использованием класса string
#include <iostream>
#include <string>  // класс string
int main()
{
   using namespace std;
   string word = "?ate";
   for (char ch = 'a'; word != "mate"; ch++)
   {
      cout << word << endl;
      word[0] = ch;
   }
   cout << "After loop ends, word is " << word << endl;
   return 0;
}
```

Вывод этой программы в точности такой же, как у программы из листинга 5.11.

Замечания по программе

В листинге 5.12 приведенное ниже проверочное условие использует операцию сравнения с объектом `string` в левой части и строкой в стиле C — в правой части:

```
word != "mate"
```

Способ перегрузки операции `!=` классом `string` позволяет применять ее, если хотя бы один из операндов является объектом `string`; второй операнд при этом может быть либо объектом `string`, либо строкой в стиле C.

Класс `string` позволяет использовать объект `string` либо в качестве одиночной сущности, как в выражениях сравнения, либо в качестве агрегатного объекта, допускающего нотацию массивов для извлечения индивидуальных символов.

Как видите, одного и того же результата можно достигнуть как с помощью строки в стиле C, так и с помощью объектов `string`, но программирование с объектами `string` проще и намного понятней.

И, наконец, в отличие от большинства циклов `for`, которые вы видели до сих пор, два последних цикла не имеют счетчиков. То есть они не выполняют блок операторов определенное количество раз. Вместо этого каждый из циклов проверяет определенное условие (равенство слову `"mate"`), которое сигнализирует о необходимости завершения. Для программ C++ более типично применять в таких случаях цикл `while`, поэтому давайте рассмотрим его в следующем разделе.

Цикл `while`

Цикл `while` — это цикл `for`, у которого удалены инициализирующая и обновляющая части; в нем имеется только проверочное условие и тело:

```
while (проверочное_условие)
   тело
```

Сначала программа вычисляет выражение *проверочное_условие* в скобках. Если выражение дает в результате `true`, программа выполняет оператор (или операторы), содержащийся в теле цикла. Как и в случае с циклом `for`, тело состоит из единственного оператора либо блока, определенного фигурными скобками. После того, как завершено выполнение тела, программа возвращается к проверочному условию и заново вычисляет его. Если условие возвращает ненулевое значение, программа снова выполняет тело. Этот цикл проверки и выполнения продолжается до тех пор, пока проверочное условие не вернет `false` (рис. 5.3). Понятно, что если вы хотите в конечном итоге прервать цикл, то в теле цикла должно происходить нечто такое, что повлияет на выражение проверочного условия. Например, цикл может увеличивать значение переменной, используемой в проверочном условии, либо читать новое значение, вводимое с клавиатуры. Подобно `for`, цикл `while` является циклом с входным условием. То есть, если проверочное условие оценивается как `false` в самом начале, то программа ни разу не выполнит тело цикла.

В листинге 5.13 представлен пример работы цикла `while`. Цикл проходит по всем символам строки и отображает их ASCII-коды. По достижении нулевого символа цикл завершается. Эта техника прохода по символам строки до нулевого ограничителя является стандартным методом обработки строк в C++. Поскольку строка содержит маркер конца, программа часто не нуждается в явной информации о длине строки.

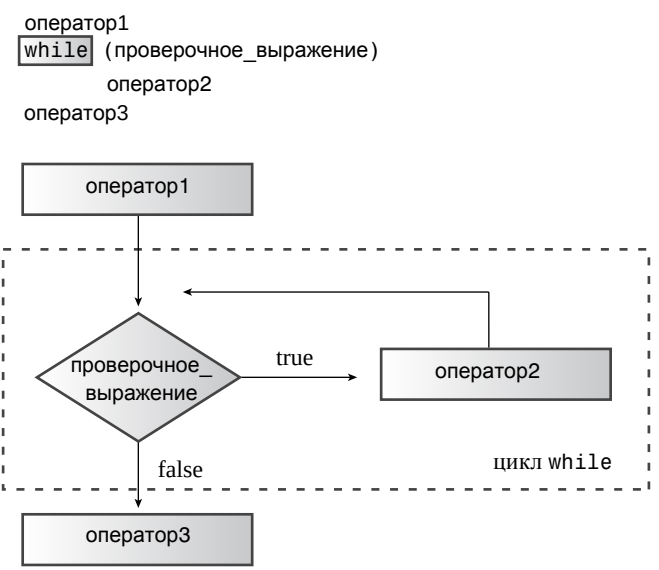


***Рис. 5.3.** Структура циклов `while`*

Листинг 5.13. `while.cpp`

```
// while.cpp -- представление цикла while
#include <iostream>
const int ArSize = 20;
int main()
{
    using namespace std;
    char name[ArSize];

    cout << "Your first name, please: ";  // ввод имени
    cin >> name;
```

```
    // Вывод имени посимвольно и в кодах ASCII
    cout << "Here is your name, verticalized and ASCIIized:\n";
    int i = 0;                  // начать с начала строки
    while (name[i] != '\0')     // обрабатывать до конца строки
    {
       cout << name[i] << ": " << int(name[i]) << endl;
       i++;                     // не забудьте этот шаг
    }
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 5.13:

```
Your first name, please: Muffy
Here is your name, verticalized and ASCIIized:
M: 77
u: 117
f: 102
f: 102
y: 121
```

Замечания по программе

Условие `while` в листинге 5.13 выглядит следующим образом:

```
while (name[i] != '\0')
```

Оно проверяет, является ли определенный символ массива нулевым. Чтобы эта проверка в конечном итоге была успешной, в теле цикла значение индекса `i` должно изменяться. Это достигается инкрементированием `i` в конце тела цикла. Если пропустить этот шаг, то цикл застрянет на одном элементе массива, печатая один и тот же символ и его код до тех пор, пока вы принудительно не завершите программу. Возможность создания бесконечной последовательности — одна из наиболее часто возникающих проблем при работе с циклами. Часто это получается именно из-за того, что вы забываете изменить в теле цикла что-то связанное с проверочным условием.

Строку с `while` можно переписать так:

```
while (name[i])
```

С этим изменением программа будет работать точно так же, как и раньше. Это объясняется тем, что `name[i]` — обычный символ, а его значение является кодом символа, который отличен от нуля, что соответствует `true`. Но когда в `name[i]` содержится нулевой символ, его код равен `0`, т.е. `false`. Такая нотация более краткая и применяется чаще, но не столь ясна, как та, что приведена в листинге 5.13. Некоторые компиляторы попроще могут порождать более эффективный код во втором случае, но более интеллектуальные компиляторы генерируют в обоих случаях одинаковый код.

Чтобы напечатать ASCII-код символа, программа использует приведение для преобразования `name[i]` в целочисленный тип. После этого `cout` печатает значение символа в виде целого числа.

В отличие от строк в стиле C, объекты класса `string` не используют нулевые символы для идентификации конца строки, поэтому модифицировать листинг 5.13 для использования `string` простой заменой массива `char` объектом `string` не получится. В главе 16 обсуждается техника, которую можно применить с объектами `string` для идентификации последнего символа.

Сравнение циклов for и while

В C++ циклы for и while, в сущности, эквивалентны. Например, следующий цикл for:

```
for (инициализирующее-выражение; проверочное-выражение; обновляющее-выражение)
{
    оператор(ы)
}
```

может быть переписан так:

```
инициализирующее-выражение;
while (проверочное-выражение)
{
    оператор(ы)
    обновляющее-выражение;
}
```

Аналогично, представленный ниже цикл while:

```
while (проверочное-выражение)
    тело
```

можно переписать так:

```
for ( ; проверочное-выражение;)
    тело
```

Цикл for требует трех выражений (или, формально, одного оператора и следующих за ним двух выражений), но они могут быть пустыми выражениями (или операторами). Обязательны только два знака точки с запятой. Кстати, пропуск проверочного выражения в цикле for трактуется как true, поэтому следующий цикл будет продолжаться бесконечно:

```
for( ; ; )
    тело
```

Поскольку циклы for и while почти эквивалентны, какой именно использовать — в основном, вопрос стиля. Есть три отличия. Одно из них, как уже упоминалось, заключается в том, что пропущенное условие проверки в цикле for интерпретируется как true. Второе отличие связано с возможностью использования оператора инициализации в цикле for для объявления переменной, которая будет локальной в цикле; в цикле while это сделать не получится. Наконец, существует небольшое отличие, когда тело цикла содержит оператор continue, который описан в главе 6. Обычно программисты применяют циклы for для циклов со счетчиками, потому что формат for позволяет держать всю необходимую информацию — начальное значение, конечное значение и метод обновления счетчика — в одном месте. Цикл while используется, когда заранее не известно, сколько раз будет выполняться цикл.

Совет

При проектировании цикла необходимо руководствоваться следующими указаниями.

- Идентифицировать условие завершения цикла.
- Инициализировать это условие перед первой проверкой.
- Обновлять условие на каждом шаге цикла, прежде чем оно будет проверено вновь.

Одним из преимуществ цикла for является то, что его структура предоставляет место для реализации всех этих требований, и это помогает помнить о них. Следует отметить, что перечисленные выше указания применимы также и к циклу while.

Плохая пунктуация

Оба цикла — `for` и `while` — имеют тело, которое состоит из одного оператора, следующего за выражениями в скобках. Как вы уже видели, этот единственный оператор может быть блоком, содержащим несколько операторов. Помните, что блок формируют фигурные скобки, а не отступы. Например, посмотрите на следующий фрагмент кода:

```
i = 0;
while (name[i] != '\0')
   cout << name[i] << endl;
   i++;
cout << "Done\n";
```

Отступ говорит о том, что автор программы, вероятно, намеревался включить оператор `i++;` в тело цикла. Но отсутствие фигурных скобок говорит компилятору, что тело цикла состоит из единственного первого оператора `cout`. Таким образом, этот цикл будет бесконечно печатать первый символ массива. Программа никогда не достигнет оператора `i++;`, потому что он находится за пределами цикла.

Следующий пример демонстрирует другую потенциальную ловушку:

```
i = 0;
while (name[i] != '\0');  // проблема кроется в точке с запятой
{
   cout << name[i] << endl;
   i++;
}
cout << "Done\n";
```

На этот раз в коде правильно размещены фигурные скобки, но перед ними находится лишняя точка с запятой. Вспомните, что точка с запятой завершает оператор, и здесь она завершает цикл `while`. Другими словами, телом цикла является *пустой оператор* — т.е. *ничего*, за которым следует точка с запятой. Все, что находится в фигурных скобках, происходит *после* завершения цикла, и никогда не будет выполнено, потому что цикл, не делающий ничего, бесконечен. Внимательно расставляйте точки с запятой.

Построение цикла задержки

Иногда возникает необходимость приостановить выполнение программы на некоторое время. Например, программа может выдавать мгновенное сообщение на экран и тут же начать делать что-то еще, до того, как вы успеете его прочитать. В этом случае есть опасность пропустить жизненно важную информацию незамеченной. Было бы гораздо удобнее, если бы в этом случае программы приостановилась на 5 секунд, прежде чем продолжать работу. Цикл `while` удобен для создания такого эффекта. С давних времен существования персональных компьютеров для приостановки выполнения программы применялся способ, заключающийся в запуске простого счетчика:

```
long wait = 0;
while (wait < 10000)
   wait++; // молча считать
```

Проблема этого подхода состоит в том, что при переходе на компьютер с другой скоростью процессора приходилось менять предельное значение счетчика. Некоторые игры, написанные для оригинального IBM PC, например, становились неуправляемо быстрыми при переносе на более производительные компьютеры.

В наши дни компилятор может даже заключить, что вполне достаточно установки `wait` в 1000 и вообще пропустить цикл. Более правильный подход предусматривает использование системных часов для организации задержки.

Библиотеки ANSI C и C++ включают функцию, которая помогает в этом. Она называется `clock()` и возвращает системное время, прошедшее с момента запуска программы. Однако с ней связано несколько сложностей. Во-первых, `clock()` не обязательно возвращает время в секундах. Во-вторых, типом возврата этой функции в одних системах может быть `long`, в других — `unsigned long`, а в третьих — еще каким-нибудь.

Заголовочный файл `ctime` (`time.h` в более старых реализациях) предлагает решение этих проблем. Во-первых, он определяет символическую константу `CLOCKS_PER_SEC`, которая содержит количество единиц системного времени, приходящихся на секунду. То есть, разделив показание системного времени на эту константу, вы получите секунды. Или же вы можете умножить секунды на `CLOCKS_PER_SEC`, чтобы получить время в системных единицах. Во-вторых, `ctime` устанавливает псевдоним `clock_t` для типа возврата `clock()`. (См. ниже врезку "Псевдонимы типов".) Это значит, что вы можете объявить переменную типа `clock_t`, и компилятор преобразует ее в `long` или `unsigned int` либо в любой другой подходящий для системы тип.

В листинге 5.14 демонстрируется использование `clock()` и `ctime` для организации цикла задержки.

Листинг 5.14. `waiting.cpp`

```
// waiting.cpp -- использование clock() в цикле временной задержки
#include <iostream>
#include <ctime>  // описывает функцию clock() и тип clock_t
int main()
{
   using namespace std;
   cout << "Enter the delay time, in seconds: ";  // ввод времени задержки в секундах
   float secs;
   cin >> secs;
   clock_t delay = secs * CLOCKS_PER_SEC;         // преобразование в тики
   cout << "starting\a\n";
   clock_t start = clock();
   while (clock() - start < delay )               // ожидание истечения времени
      ;  // обратите внимание на точку с запятой
   cout << "done \a\n";
   return 0;
}
```

Подсчитывая время задержки в системных единицах вместо секунд, программа в листинге 5.14 избегает необходимости преобразования времени в секунды на каждом шаге цикла.

Псевдонимы типов

В C++ предусмотрены два способа установки нового имени в качестве псевдонима для типа. Один — через препроцессор:

```
#define BYTE char // препроцессор заменяет BYTE на char
```

Препроцессор затем заменяет все вхождения `BYTE` на `char` во время компиляции программы, таким образом, рассматривая `BYTE` как псевдоним `char`.

Второй способ заключается в применении ключевого слова C (или C++) `typedef` для создания псевдонима. Например, чтобы объявить `byte` псевдонимом `char`, вы поступаете следующим образом:

```
typedef char byte; // делает byte псевдонимом char
```

Вот обобщенная форма:

```
typedef имяТипа имяПсевдонима;
```

Другими словами, если вы хотите, чтобы *имяПсевдонима* был псевдонимом для определенного типа, вы объявляете *имяПсевдонима*, как если бы он был переменной этого типа, и предваряете объявление ключевым словом `typedef`.

Например, чтобы сделать `byte_pointer` псевдонимом указателя на `char`, вы должны объявить `byte_pointer` как указатель на `char` и поставить впереди `typedef`:

```
typedef char * byte_pointer; // указатель на тип char
```

Можете попробовать что-то подобное реализовать через `#define`, но это не работает, когда объявляется список переменных. Например, рассмотрим следующее:

```
#define FLOAT_POINTER float *
FLOAT_POINTER pa, pb;
```

Препроцессор выполнит подстановку и превратит это объявление в следующее:

```
float * pa, pb; // pa – указатель на float, а pb – просто float
```

Подход `typedef` лишен этой проблемы. Способность объявлять более сложные псевдонимы типов делает применение `typedef` более предпочтительным выбором по сравнению `#define`, а иногда и единственно возможным.

Обратите внимание, что ключевое слово `typedef` не создает нового типа. Оно просто назначает существующему типу новое имя. Если вы объявляете `word` как псевдоним `int`, `cout` трактует значения типа `word`, как если бы они были типа `int`.

Цикл do while

К этому моменту вы ознакомились с двумя циклами — `for` и `while`. Третьим циклом в C++ является `do while`. Он отличается от двух других тем, что осуществляет проверку на выходе. Это значит, что такой цикл вида "кто его знает" сначала выполнит свое тело и только потом оценит проверочное условие, чтобы узнать, нужно ли продолжать дальше. Если условие оценивается как `false`, цикл завершается; в противном случае выполняется новый шаг с последующей проверкой условия. Такой цикл всегда выполняется, как минимум, один раз, потому что поток управления программы проходит через его тело до того, как достигает проверочного условия. Синтаксис цикла `do while` показан ниже:

```
do
   тело
while (проверочное-выражение);
```

Часть *тело* может быть единственным оператором либо блоком операторов, заключенным в фигурные скобки. На рис. 5.4 показан поток управления в цикле `do while`.

Обычно цикл с проверкой на входе — лучший выбор, нежели цикл с проверкой на выходе, т.к. проверка выполняется до его запуска. Например, предположим, что в листинге 5.13 использовался бы `do while` вместо `while`. В этом случае цикл должен будет печатать нулевой символ и его код, прежде чем обнаружит, что он уже достиг конца строки. Но все же иногда проверка в стиле `do while` имеет смысл. Например, если вы запрашиваете пользовательский ввод, то программа должна получить его, а только затем проверить. В листинге 5.15 показано, как можно использовать `do while` в такой ситуации.

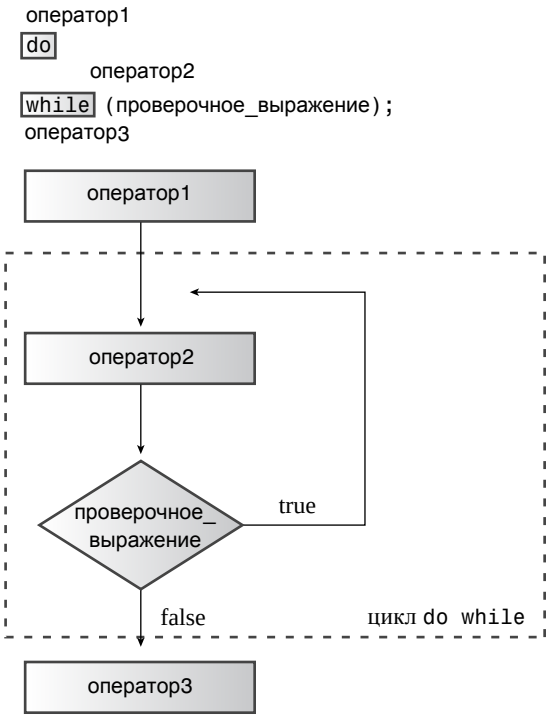


Рис 5.4. Структура циклов `do while`

Листинг 5.15. `dowhile.cpp`

```
// dowhile.cpp -- цикл с проверкой на выходе
#include <iostream>
int main()
{
   using namespace std;
   int n;
   cout << "Enter numbers in the range 1-10 to find ";
   cout << "my favorite number\n"; // запрос на ввод любимого числа из диапазона 1-10
   do
   {
      cin >> n;                          // выполнить тело
   } while (n != 7);                     // затем проверить
   cout << "Yes, 7 is my favorite.\n" ; // любимое число - 7
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 5.15:

```
Enter numbers in the range 1-10 to find my favorite number
9
4
7
Yes, 7 is my favorite.
```

Странные циклы for

Не очень часто, но иногда вам может встретиться код, который представляет нечто такое:

```
int I = 0;
for(;;) // иногда называется "бесконечным циклом"
{
   I++;
   // делать что-то...
   if (30 >= I) break; // операторы if и break (см. главу 6)
}
```

А вот другой вариант:

```
int I = 0;
for(;;I++)
{
   if (30 >= I) break;
   // делать что-то...
}
```

Этот код полагается на тот факт, что пустое проверочное условие в цикле `for` трактуется как истинное. Ни один из этих примеров не является простым для чтения, и ни один из них не стоит использовать в качестве общей модели при написании циклов. Функциональность первого примера может быть выражена яснее с помощью цикла `do while`:

```
int I = 0;
do {
   I++;
   // делать что-то...
while (30 < I);
```

Аналогично, второй пример может быть выражен более ясно посредством цикла `while`:

```
while (I < 30)
{
   // делать что-то...
   I++;
}
```

В общем случае написание чистого и хорошо понятного кода — более важная цель, нежели демонстрация умения применять малоизвестные средства языка.

Цикл `for`, основанный на диапазоне (C++11)

В C++11 была добавлена новая форма цикла, которая называется циклом `for`, *основанным на диапазоне*. Она упрощает одну общую задачу цикла — делать что-то с каждым элементом массива или, в более общем случае, с одним из контейнерных классов, таким как `vector` или `array`.

Ниже показан пример:

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (double x : prices)
   cout << x << std::endl;
```

Здесь `x` изначально представляет первый член массива `prices`. После отображения первого элемента цикл затем проходит по `x` для представления оставшихся элементов массива, так что код выведет все пять членов, по одному в строке. Короче говоря, этот цикл отображает все значения, включенные в диапазон массива.

Чтобы изменить значения в массиве, понадобится применить другой синтаксис для переменной цикла:

```
for (double &x : prices)
   x = x * 0.80; // скидка 20%
```

Символ & идентифицирует x как ссылочную переменную; эта тема раскрывается в главе 8. Здесь важно то, что такая форма объявления позволяет последующему коду изменять содержимое массива, тогда как первая форма не разрешает этого.

Цикл for, основанный на диапазоне также может использоваться со списками инициализации:

```
for (int x : {3, 5, 2, 8, 6})
   cout << x << " ";
cout << '\n';
```

Однако этот цикл, скорее всего, наиболее часто будет применяться с различными шаблонными классами контейнеров, которые рассматриваются в главе 16.

Циклы и текстовый ввод

Теперь, когда вы узнали, как работают циклы, давайте рассмотрим одну из наиболее часто встречающихся и важных задач, выполняемых циклами: чтение текстового ввода из файла или с клавиатуры символ за символом.

Например, вам может понадобиться написать программу, которая подсчитывает количество символов, строк и слов во входном потоке. Традиционно в C++, как и в C, для решения задач подобного рода используется цикл while. Давайте посмотрим, как это делается. Если вы уже знаете C, не переходите слишком быстро к следующему разделу. Хотя цикл C++ while — точно такой же, как в C, средства ввода-вывода в C++ отличаются. Это может придать циклу C++ несколько другой вид, чем у цикла C. Фактически, объект cin поддерживает три разных режима односимвольного ввода, каждый с собственным пользовательским интерфейсом. Ниже будет показано, как использовать эти варианты в циклах while.

Применение для ввода простого cin

Если программа собирается использовать цикл для чтения текстового ввода с клавиатуры, она должна каким-то образом узнать, когда следует остановиться. Как программа узнает об этом? Один из способов заключается в использовании некоторого специального символа, иногда называемого *сигнальным символом* в качестве сигнала останова. Например, листинг 5.16 прекращает чтение ввода, когда программа встречает символ #. Программа подсчитывает количество прочитанных символов и отображает их, т.е. повторно выводит прочитанные символы. (Нажатие клавиши на клавиатуре не приводит к автоматическому помещению соответствующего символа на экран; программы должны самостоятельно выполнять всю нудную работу по отображению введенного символа. Обычно эту задачу обрабатывает операционная система. В данном случае отображать введенные символы будет как операционная система, так и тестовая программа.) По завершении работы программа выдаст отчет об общем количестве обработанных символов. Исходный код этой программы приведен в листинге 5.16.

Листинг 5.16. textin1.cpp

```
// textin1.cpp -- чтение символов в цикле while
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;              // использование базового ввода
   cout << "Enter characters; enter # to quit:\n";
   cin >> ch;                  // получение символа
   while (ch != '#')           // проверка символа
   {
      cout << ch;              // отображение символа
      ++count;                 // подсчет символа
      cin >> ch;               // получение следующего символа
   }
   cout << endl << count << " characters read\n";
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 5.16:

```
Enter characters; enter # to quit:
see ken run#really fast
seekenrun
9 characters read
```

Как видите, пробельные символы из введенной строки в выводе отсутствуют.

Замечания по программе

Обратите внимание на структуру программы в листинге 5.16. Программа читает первый введенный символ до входа в цикл. Таким образом, первый символ может быть проверен, когда программа достигает оператора цикла. Это важно, потому что первым символом может сразу оказаться #. Поскольку `textin1.cpp` использует цикл с проверкой на входе, в этом случае программа корректно пропустит весь цикл. А поскольку переменной `count` было предварительно присвоено значение `0`, `count` будет содержать правильное значение.

Предположим, что первый прочитанный символ отличается от #. В этом случае программа входит в цикл, отображает символ, увеличивает значение счетчика `count` и читает следующий символ. Последний шаг жизненно важен. Без него цикл бесконечно обрабатывал бы первый введенный символ. Но благодаря этому последнему шагу, программа может перейти к следующему символу.

Обратите внимание, что структура цикла следует упомянутым ранее правилам. Условие, прекращающее выполнение цикла — когда последним прочитанным символом является #. Это условие инициализируется первым чтением символа перед входом в цикл. Условие обновляется чтением следующего символа в конце тела цикла.

Все это звучит вполне разумно. Но почему же программа не выводит пробелы? В этом виноват объект `cin`. Когда он читает значения типа `char`, как и при чтении других базовых типов, он пропускает пробелы и символы новой строки. Эти символы не отображаются и не могут быть подсчитаны.

Чтобы еще более усложнить ситуацию, сообщим, что ввод в `cin` буферизуется. Это значит, что вводимые символы не попадут в программу до тех пор, пока не будет нажата клавиша <Enter>. Вот почему программа из листинга 5.16 позволяет печатать символы и после #. После нажатия <Enter> вся последовательность символов передается в программу, но программа прекращает обработку ввода после прочтения #.

Спасение в виде cin.get(char)

Обычно программы, принимающие ввод символ за символом, должны обрабатывать каждый введенный символ, включая пробелы, знаки табуляции и символы новой строки. Класс istream (определенный в iostream), к которому относится объект cin, включает функцию-член, которая соответствует этому требованию. В частности, функция cin.get(ch) читает из ввода следующий символ, даже если это пробел, и присваивает его переменной ch. Заменив cin >> ch вызовом этой функции, вы можете устранить недостатки программы из листинга 5.16. В листинге 5.17 показан переработанный код.

Листинг 5.17. textin2.cpp

```
// textin2.cpp -- использование cin.get(char)
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;
   cout << "Enter characters; enter # to quit:\n";
   cin.get(ch);                // использование функции cin.get(ch)
   while (ch != '#')
   {
      cout << ch;
      ++count;
      cin.get(ch);             // использование ее снова
   }
   cout << endl << count << " characters read\n";
   return 0;
}
```

Ниже приведен пример выполнения версии программы из листинга 5.17:

```
Enter characters; enter # to quit:
Did you use a #2 pencil?
Did you use a
14 characters read
```

Теперь программа отображает и подсчитывает все символы, включая пробелы. Ввод по-прежнему буферизуется, поэтому по-прежнему можно ввести больше информации, чем на самом деле попадет в программу.

Если вы знакомы с языком C, эта программа может показаться опасно ошибочной. Вызов cin.get(ch) помещает значение в переменную ch, а это означает, что она изменяет значение переданной переменной. В C вы должны передавать адрес переменной функции, если хотите, чтобы она изменила ее значение. Но при вызове cin.get() в листинге 5.17 передается ch, а не &ch. В языке C подобный код не заработает. В C++ он может работать, если функция объявляет свой аргумент как *ссылку*. Это — новое средство C++. В заголовочном файле iostream аргумент cin.get(ch) объявлен со ссылочным типом, поэтому данная функция может изменять значение своего аргумента. Более подробно об этом вы узнаете в главе 8. А пока знатоки C могут расслабиться; обычно аргументы, передаваемые в C++, работают точно так же, как и в C. Но только не в случае cin.get(ch).

Выбор используемой версии cin.get()

Листинг 4.5 в главе 4 содержит следующий код:

```
char name[ArSize];
...
cout << "Enter your name: \n";
cin.get(name, ArSize).get();
```

Последняя строка эквивалентна следующим последовательным вызовам функции:

```
cin.get(name, ArSize);
cin.get();
```

Одна версия `cin.get()` принимает два аргумента: имя массива, который представляет адрес строки (формально `char*`), и `ArSize`, который является целым типа `int`. (Вспомните, что имя массива — это адрес его первого элемента, поэтому имя символьного массива имеет тип `char*`.) Затем программа использует `cin.get()` без аргументов. Но в последнем примере функция `cin.get()` применялась следующим образом:

```
char ch;
cin.get(ch);
```

На этот раз `cin.get()` принимает один аргумент, и его типом является `char`.

И здесь опять приходит время тем, кто знаком с языком C, прийти в замешательство. В языке C, если функция принимает в качестве аргументов указатель на `char` и `int`, не получится с таким же успехом ее использовать с одним аргументом, да еще и другого типа. Но с C++ такое возможно, потому что этот язык поддерживает средство ООП под названием *перегрузка функций*. Перегрузка функций позволяет создавать разные функции с одним и тем же именем, при условии, что списки их аргументов отличаются. К примеру, если вы используете `cin.get(name, ArSize)` в C++, компилятор находит ту версию `cin.get()`, которая принимает аргументы `char*` и `int`. Но если вы применяете `cin.get(ch)`, то компилятор найдет версию `cin.get()`, которая принимает единственный аргумент типа `char`. И, наконец, если код не передает никаких аргументов, то компилятор выберет версию `cin.get()` без аргументов. Перегрузка функций позволяет использовать одно и то же имя для взаимосвязанных функций, выполняющих одну и ту же задачу разными способами или с разными типами данных. Это — еще одна тема, знакомство с которой ожидает вас в главе 8. А пока вы можете привыкать к перегрузке функций, используя примеры `get()`, поставляемые классом `istream`. Чтобы отличать друг от друга разные версии одной функции, при упоминании их мы будем указывать список аргументов. То есть `cin.get()` будет означать версию, не принимающую аргументов, а `cin.get(char)` — версию с одним аргументом.

Условие конца файла

Как показано в листинге 5.17, применение символа вроде # для обозначения конца в не всегда подходит, потому что такой ввод может быть частью совершенно легитимного ввода. То же самое верно и для любого другого символа, такого как `@` или `%`. Если ввод поступает из файла, вы можете задействовать более мощный прием — обнаружение конца файла (end-of-file — EOF). Средства ввода C++ взаимодействуют с операционной системой для обнаружения момента достижения конца файла, и предоставляют эту информацию программе.

На первый взгляд чтение информации из файла имеет мало общего с `cin` и клавиатурным вводом, но между ними существуют две связи. Во-первых, многие операционные системы, включая Unix, Linux и режим командной строки Windows, поддерживают *перенаправление*, что позволяет легко подставлять файл вместо клавиатурного ввода. Например, предположим, что в среде Windows имеется исполняемая программа `gofish.exe` и текстовый файл по имени `fishtale`. В этом случае вы можете вести следующую команду в командной строке:

```
gofish <fishtale
```

Это заставит программу принять ввод из файла `fishtale` вместо клавиатуры. Символом < обозначается операция перенаправления, как в Unix, так и в режиме командной строки Windows.

Во-вторых, многие операционные системы позволяют эмулировать условие EOF с клавиатуры. В Unix для этого необходимо нажать клавиатурную комбинацию <Ctrl+D> в начале строки. В режиме командной строки Windows для этого потребуется нажать <Ctrl+Z> и после этого <Enter> в любом месте строки. Некоторые реализации C++ поддерживают подобное поведение, даже если его не поддерживает операционная система. Концепция EOF для клавиатурного ввода в действительности унаследована от сред командной строки. Однако Symantec C++ для Mac имитирует Unix и распознает <Ctrl+D> как эмуляцию EOF. Версия Metrowerks Codewarrior распознает <Ctrl+Z> в средах Macintosh и Windows. Версии Microsoft Visual C++, Borland C++ 5.5 и GNU C++ для ПК распознают комбинацию <Ctrl+Z>, когда она встречается в начале строки, но требуют последующего нажатия <Enter>. Короче говоря, многие среды программирования для ПК распознают комбинацию <Ctrl+Z> как эмуляцию EOF, но конкретные детали (в любом месте строки или же только в начале, требуется последующее нажатие <Enter> или нет) могут варьироваться.

Если в среде программирования предусмотрена проверка EOF, вы можете использовать программу, подобную приведенной в листинге 5.17, с перенаправленными файлами либо с клавиатурным вводом, в котором эмулируется EOF. Это выглядит удобным, поэтому давайте посмотрим, как такое делается.

Когда объект `cin` обнаруживает EOF, он устанавливает два бита (`eofbit` и `failbit`) в 1. Для проверки состояния `eofbit` можно использовать функцию-член по имени `eof()`; вызов `cin.eof()` возвращает булевское значение `true`, если EOF был обнаружен, и `false` — в противоположном случае. Аналогично, функция-член `fail()` возвращает `true`, если `eofbit` или `failbit` установлены в 1.

Обратите внимание, что методы `eof()` и `fail()` сообщают результат самой последней попытки чтения; т.е. они сообщают о прошлом, а не заглядывают в будущее. Поэтому проверки `cin.eof()` и `cin.fail()` должны всегда следовать за попытками чтения. Программа в листинге 5.18 отражает этот факт. В ней используется `fail()` вместо `eof()`, потому что первый из этих методов работает в более широком диапазоне реализаций.

На заметку!

Некоторые системы не поддерживают эмуляцию EOF из клавиатуры. Другие ее поддерживают, но не лучшим образом. Если вы использовали `cin.get()` для приостановки вывода на экран, чтобы можно было прочитать его, то здесь это не работает, потому что обнаружение EOF отключает дальнейшие попытки чтения ввода. Однако можно организовать цикл задержки, подобный использованному в листинге 5.14, чтобы на время оставить экран видимым. Или же можно применить `cin.clear()`, как будет показано в главах 6 и 17, чтобы сбросить поток ввода.

Листинг 5.18. textin3.cpp

```
// textin3.cpp — чтение символов до конца файла
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int count = 0;
   cin.get(ch);                          // попытка чтения символа
   while (cin.fail() == false)           // проверка на EOF
   {
      cout << ch;                        // отображение символа
      ++count;
      cin.get(ch);                       // попытка чтения следующего символа
   }
   cout << endl << count << " characters read\n";
   return 0;
}
```

Ниже приведен пример выполнения программы из листинга 5.18:

```
The green bird sings in the winter.<Enter>
The green bird sings in the winter.
Yes, but the crow flies in the dawn.<Enter>
Yes, but the crow flies in the dawn.
<Ctrl+Z><Enter>
73 characters read
```

Поскольку эта программа запускалась в системе Windows 7, для эмуляции условия EOF нажималась клавиатурная комбинация <Ctrl+Z> и затем <Enter>. Пользователи Unix и Linux должны вместо этого нажимать <Ctrl+D>. Обратите внимание, что в Unix и Unix-подобных системах, включая Linux и Cygwin, нажатие <Ctrl+Z> приостанавливает выполнение программы; возобновить ее выполнение можно с помощью команды `fg`.

За счет применения перенаправления программу из листинга 5.18 можно использовать для отображения текстового файла и подсчета количества содержащихся в нем символов. На этот раз мы применим ее для чтения, отображения и подсчета символов в файле, содержащем две строки, в системе Unix (`$` — это приглашение командной строки Unix):

```
$ textin3 < stuff
I am a Unix file. I am proud
to be a Unix file.
48 characters read
$
```

Признак EOF завершает ввод

Вспомните, что когда метод `cin` обнаруживает EOF, он устанавливает флаг в объекте `cin`, обозначающий условие EOF. Когда установлен этот флаг, `cin` не читает больше никакого ввода, и последующие вызовы `cin` не дают никакого эффекта. Для файлового ввода это имеет смысл, поскольку вы не можете читать ничего за концом файла. Однако при клавиатурном вводе вы можете эмулировать EOF для прерывания цикла, но захотите позже продолжать вводить информацию. Метод `cin.clear()` очищает флаг EOF и позволяет продолжить обработку ввода. В главе 17 это обсуждается более подробно. Однако имейте в виду, что на некоторых системах нажатие

<Ctrl+Z> полностью прекращает ввод и вывод, несмотря на возможность их восстановления с помощью `cin.clear()`.

Общие идиомы символьного ввода

Ниже представлена структура цикла, предназначенного для чтения текста по одному символу вплоть до получения EOF:

```
cin.get(ch);                    // попытка чтения символа
while (cin.fail() == false)     // проверка на EOF
{
   ...                          // делать что-то полезное
   cin.get(ch);                 // попытка чтения следующего символа
}
```

Этот код можно несколько сократить. В главе 6 будет представлена операция `!`, которая обращает `true` в `false` и наоборот. С ее применением проверочное условие `while` можно переписать следующим образом:

```
while (!cin.fail())             // пока ввод не даст сбой
```

Возвращаемое значение `cin.get(char)` — это сам объект `cin`. Однако в классе `istream` предусмотрена функция, которая преобразует такой объект `istream`, как `cin`, в булевское значение; эта функция преобразования неявно вызывается, когда `cin` появляется в выражениях, где ожидается значение типа `bool`, например, в проверочном условии цикла `while`. Более того, это значение `bool` равно `true`, только если последняя попытка чтения завершилась успешно, в противном случае оно равно `false`. Это значит, что проверочное условие цикла `while` можно переписать так:

```
while (cin)                     // пока ввод успешен
```

Эта проверка несколько более обширна, чем применение `!cin.fail()` или `!cin.eof()`, потому что обнаруживает также и ряд других причин сбоев, таких как отказ диска.

И, наконец, поскольку возвращаемое значение `cin.get(ch)` — это сам объект `cin`, вы можете сократить цикл до следующего вида:

```
while (cin.get(ch))             // пока ввод успешен
{
   ...                          // делать что-то полезное
}
```

Здесь `cin.get(char)` вызывается лишь однажды, в составе проверочного условия, а не дважды — один раз перед циклом и один — в конце тела цикла, как было раньше. Чтобы оценить проверочное условие цикла, программа сначала должна выполнить вызов `cin.get(ch)`, в случае успеха которого введенный символ будет помещен в `ch`. Затем программа получает возвращаемое значение этого вызова — сам объект `cin`. Затем она применяет преобразование `cin` в значение `bool`, которое равно `true`, если ввод успешно отработал, и `false` — в противном случае. Все три рекомендации — идентификация условия завершения, инициализация этого условия и его обновление — оказались упакованными в одно проверочное условие цикла.

Еще одна версия `cin.get()`

Ностальгирующие пользователи C могут тосковать по функциям ввода-вывода `getchar()` и `putchar()`. Эти функции могут быть доступны и в C++, если вы в них нуждаетесь. Для этого понадобится включить заголовочный файл `stdio.h`, как это делалось в программах C (либо обратиться к его более новой версии `cstdio`).

Или же вы можете применять функции классов istream и ostream, которые работают практически так же. Давайте теперь рассмотрим и этот подход.

Функция-член cin.get() без аргументов возвращает следующий символ ввода. То есть, вы можете использовать ее следующим образом:

```
ch = cin.get();
```

(Вспомните, что cin.get(ch) возвращает объект, а не прочитанный символ.) Эта функция работает почти так же, как getchar() в языке C, возвращая код символа в виде значения типа int. Аналогично вы можете использовать функцию cout.put() (см. главу 3) для отображения символа:

```
cout.put(ch);
```

Она работает очень похоже на putchar() в C, но с одним отличием — аргумент должен быть типа char, а не int.

На заметку!

Изначально функция-член put() имела единственный прототип — put(char). Ей можно передавать аргумент int, который затем приводится к char. В стандарте также утвержден один прототип. Однако некоторые реализации C++ предлагают три прототипа: put(char), put(signed char) и put(unsigned char). Вызов put() с аргументом типа int в этих реализациях генерирует сообщение об ошибке, поскольку преобразование в int допускают более одного варианта. Явное приведение типа, такое как cin.put(char(ch)), позволяет передавать int.

Чтобы успешно применять cin.get(), вы должны знать, как эта версия функции обрабатывает условие EOF. Когда функция достигает EOF, не остается символов, которые должны быть возвращены. Вместо этого cin.get() возвращает специальное значение, представленное символической константой EOF. Эта константа определена в заголовочном файле iostream. Значение EOF должно отличаться от любого допустимого значения символа, чтобы программа не могла спутать EOF с обычным символом. Обычно EOF определяется как -1, т.к. ни один из символов не имеет ASCII-кода, равного -1. Однако вам не обязательно знать действительное значение. В программах вы просто используете EOF. Например, основная часть листинга 5.18 выглядит следующим образом:

```
char ch;
cin.get(ch);
while (cin.fail() == false) // проверка на EOF
{
   cout << ch;
   ++count;
   cin.get(ch);
}
```

Вы можете использовать int ch, заменить cin.get(ch) на cin.get(), заменить cout на cout.put() и заменить cin.fail() проверкой на EOF:

```
int ch;                    // для обеспечения совместимости со значением EOF
ch = cin.get();
while (ch != EOF)
{
   cout.put(ch);           // cout.put(char(ch)) для некоторых реализаций
   ++count;
   ch = cin.get();
}
```

Если ch является символом, цикл отображает его. Если же ch равно EOF, цикл завершается.

На заметку

Вы должны понимать, что EOF не представляет символ в потоке ввода. Это просто сигнал о том, что символов больше нет.

Существует один тонкий, но важный момент, касающийся применения cin.get(), о котором еще не было сказано. Поскольку EOF представляет значение, находящееся вне допустимых кодов символов, может случиться, что оно будет не совместимо с типом char. Например, в некоторых системах тип char является беззнаковым, поэтому переменная типа char никогда не получит обычного значения EOF, равного -1. По этой причине, если вы используете cin.get() (без аргументов) и проверяете возврат на EOF, то должны присваивать возвращенное значение int, а не char. Кроме того, если вы объявите ch типа int вместо char, то, возможно, вам придется выполнить приведение к типу char при его отображении.

В листинге 5.19 представлен подход cin.get() в новой версии программы из листинга 5.18. Он также сокращает код за счет комбинации символьного ввода с проверочным условием цикла while.

Листинг 5.19. textin4.cpp

```
// textin4.cpp -- чтение символов с помощью cin.get()
#include <iostream>
int main(void)
{
    using namespace std;
    int ch;                                 // должно быть типа int, а не char
    int count = 0;
    while ((ch = cin.get()) != EOF)         // проверка конца файла
    {
        cout.put(char(ch));
        ++count;
    }
    cout << endl << count << " characters read\n";
    return 0;
}
```

На заметку!

Определенные системы, которые либо не поддерживают эмуляцию EOF с клавиатуры, либо поддерживают ее не лучшим образом, могут помешать выполнению примера 5.19 так, как описано. Если вы используете cin.get() для приостановки вывода на экран с целью прочтения, то это не будет работать, потому что обнаружение EOF отключает дальнейшие попытки чтения ввода. Однако вы можете использовать цикл задержки вроде продемонстрированного в листинге 5.14, чтобы на время сохранить экран видимым.

Ниже приведен пример выполнения программы из листинга 5.19:

```
The sullen mackerel sulks in the shadowy shallows.<Enter>
The sullen mackerel sulks in the shadowy shallows.
Yes, but the blue bird of happiness harbors secrets.<Enter>
Yes, but the blue bird of happiness harbors secrets.
<Ctrl+Z><Enter>
104 characters read
```

Давайте проанализируем следующее условие цикла:

```
while ((ch = cin.get()) != EOF)
```

Скобки, в которые заключено подвыражение `ch = cin.get()`, заставляют программу вычислить его первым. Чтобы выполнить вычисление, программа сначала вызывает функцию `cin.get()`. Затем она присваивает возвращенное значение функции переменной `ch`. Поскольку значением оператора присваивания является значение левого операнда, то полное подвыражение сводится к значению `ch`. Если это значение равно `EOF`, цикл завершается, в противном случае — продолжается. Проверочному условию нужны все эти скобки. Предположим, что вы уберете пару скобок:

```
while (ch = cin.get() != EOF)
```

Операция `!=` имеет более высокий приоритет, чем `=`, поэтому сначала программа сравнит возвращаемое значение `cin.get()` с `EOF`. Сравнение даст в результате `true` или `false`; значение `bool` преобразуется в `0` или `1`, и это присваивается `ch`.

С другой стороны, применение `cin.get(ch)` (с аргументом) для ввода не создает никаких проблем, связанных с типом. Вспомните, что `cin.get(char)` не присваивает специального значения `ch` по достижении EOF. Фактически, в этом случае она не присваивает `ch` ничего. Переменной `ch` никогда не приходится хранить значения, отличные от символьных.

В табл. 5.3 показаны различия между `cin.get(char)` и `cin.get()`.

Таблица 5.3. Сравнение `cin.get(char)` и `cin.get()`

Свойство	`cin.get(char)`	`cin.get()`
Метод доставки вводимого символа	Присваивание аргументу `ch`	Возвращаемое значение
Возвращаемое значение функции при символьном вводе	Объект класса `istream` (`true` после преобразования к `bool`)	Код символа как значение типа `int`
Возвращаемое значение функции при EOF	Объект класса `istream` (`false` после преобразования к `bool`)	EOF

Итак, что вы должны использовать — `cin.get()` или `cin.get(char)`? Форма с символьным аргументом более полно интегрирована в объектный подход, поскольку ее возвращаемым значением является объект `istream`. Это значит, например, что вы можете связывать вызовы в цепочку. Скажем, приведенный ниже код означает чтение следующего входящего символа в `ch1` и затем следующего — в `ch2`:

```
cin.get(ch1).get(ch2);
```

Это работает, потому что вызов функции `cin.get(ch1)` возвращает объект `cin`, который затем работает как объект, к которому присоединен следующий вызов `get(ch2)`.

Возможно, основное назначение формы `get()` — обеспечить возможность быстрого чернового перехода от функций `getchar()` и `putchar()` из `stdio.h` к методам класса `iostream` — `cin.get()` и `cout.put()`. Вы просто заменяете один заголовочный файл другим и проводите глобальную замену `getchar()` и `putchar()` на эквивалентные им методы. (Если старый код использует переменную типа `int` для ввода, то вы должны будете также внести соответствующие изменения, если ваша реализация поддерживает несколько прототипов `put()`.)

Вложенные циклы и двумерные массивы

Ранее в этой главе вы уже видели, что цикл `for` — это естественный инструмент для обработки массивов. Теперь мы сделаем еще один шаг и посмотрим, как цикл `for`, внутри которого находится еще один цикл `for` (вложенные циклы), может применяться для обработки двумерных массивов.

Для начала давайте посмотрим, что собой представляет двумерный массив. Массивы, которые мы использовали до сих пор в этой главе, относятся к *одномерным массивам*, потому что каждый из них можно визуализировать как одну строку данных. Двумерный массив можно визуально представить в виде таблицы, состоящей из строк и столбцов. Двумерный массив можно использовать, например, для представления квартальных показателей по продажам в разных регионах, причем каждая строка данных будет представлять один регион. Или же можно применить двумерный массив для представления положения робота на поле компьютерной игры.

В C++ не предусмотрен специальный тип представления двумерных массивов. Вместо этого создается массив, каждый элемент которого является массивом.

Например, предположим, что требуется сохранить данные о максимальной температуре в четырех городах за четырехлетний период. В этом случае можно объявить массив следующим образом:

```
int maxtemps[4][5];
```

Это объявление означает, что `maxterms` является массивом из четырех элементов. Каждый из этих элементов сам является массивом из пяти элементов (рис. 5.5). Массив `maxterms` можно интерпретировать как представление четырех строк, по пять значений температуры в каждой.

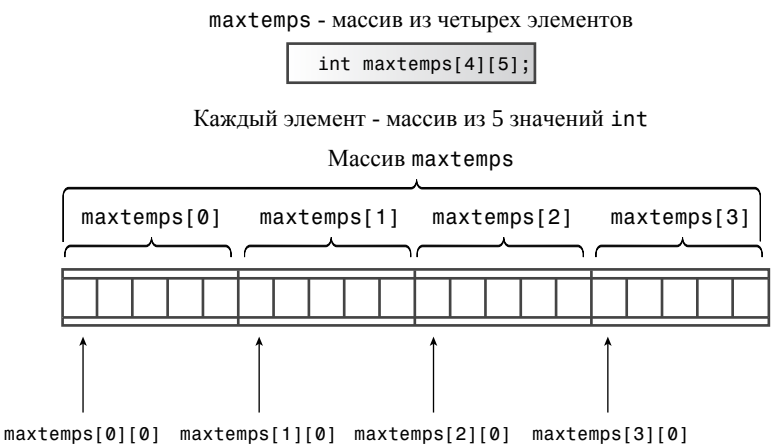


***Рис. 5.5.** Массив массивов*

Выражение `maxtemps[0]` означает первый элемент массива `maxtemps`. Таким образом, `maxtemps[0]` — сам по себе массив из пяти `int`. Первым элементом массива `maxtemps[0]` является `maxtemps[0][0]`, и этот элемент имеет тип `int`. Таким образом, для доступа к элементам `int` должны использоваться два индекса. Первый индекс можно представлять как строку таблицы, а второй — как ее столбец (рис. 5.6).

```
int maxtemps[4][5];
```

Массив maxtemps, представленный в виде таблицы

	0	1	2	3	4
maxtemps[0]	0	maxtemps[0][0]	maxtemps[0][1]	maxtemps[0][2]	maxtemps[0][3]	maxtemps[0][4]
maxtemps[1]	1	maxtemps[1][0]	maxtemps[1][1]	maxtemps[1][2]	maxtemps[1][3]	maxtemps[1][4]
maxtemps[2]	2	maxtemps[2][0]	maxtemps[2][1]	maxtemps[2][2]	maxtemps[2][3]	maxtemps[2][4]
maxtemps[3]	3	maxtemps[3][0]	maxtemps[3][1]	maxtemps[3][2]	maxtemps[3][3]	maxtemps[3][4]

Рис. 5.6. Доступ к элементам массива по индексам

Предположим, что требуется распечатать все содержимое массива. В этом случае можно использовать цикл `for` для прохода по строкам и второй вложенный цикл `for` — для прохода по столбцам:

```
for (int row = 0; row < 4; row++)
{
   for (int col = 0; col < 5; ++col)
      cout << maxtemps[row][col] << "\t";
   cout << endl;
}
```

Для каждого значения `row` вложенный цикл `for` проходит по значениям `col`. Этот пример печатает символ табуляции (`\t` в нотации управляющих символов C++) после каждого значения и символ новой строки после каждой полной строки.

Инициализация двумерного массива

При создании двумерного массива имеется возможность инициализировать каждый его элемент. Прием основан на способе инициализации одномерного массива. Вспомните, что это делается указанием разделенного запятыми списка элементов, заключенного в фигурные скобки:

```
// Инициализация одномерного массива
int btus[5] = {23, 26, 24, 31, 28};
```

В двумерном массиве каждый элемент сам по себе является массивом, поэтому инициализировать каждый элемент можно так, как показано в предыдущем примере. То есть инициализация состоит из разделенной запятыми последовательности одномерных инициализаций, каждая из которых заключена в фигурные скобки:

```
int maxtemps[4][5] =                // двумерный массив
{
   {96, 100, 87, 101, 105},         // значения для maxtemps[0]
   {96, 98, 91, 107, 104},          // значения для maxtemps[1]
   {97, 101, 93, 108, 107},         // значения для maxtemps[2]
   {98, 103, 95, 109, 108}          // значения для maxtemps[3]
};
```

Массив `maxtemps` содержит четыре строки по пять чисел в каждой.

Выражение `{96, 100, 87, 101, 105}` инициализирует первую строку, представляемую как `maxtemps[0]`. Стиль размещения каждой строки данных в отдельной строке кода улучшает читабельность.

Использование двумерного массива

Листинг 5.20 демонстрирует в одной программе инициализацию двумерного массива и проход по его элементам во вложенном цикле. На этот раз порядок циклов в программе изменен на противоположный, с помещением прохода по столбцам (индекс города) во внешний цикл, а прохода по строкам (индекс года) — во внутренний цикл. К тому же здесь используется общепринятая в C++ практика инициализации массива указателей набором строковых констант. То есть `cities` объявлен как массив указателей на `char`. Это делает каждый его элемент, такой как `cities[0]`, указателем на `char`, который может быть инициализирован адресом строки. Программа инициализирует `cities[0]` адресом строки `"Gribble City"` и т.д. Таким образом, массив указателей ведет себя как массив строк.

Листинг 5.20. `nested.cpp`

```
// nested.cpp -- вложенные циклы и двумерный массив
#include <iostream>
const int Cities = 5;
const int Years = 4;
int main()
{
   using namespace std;
   const char * cities[Cities] =        // массив указателей на 5 строк
   {
      "Gribble City",
      "Gribbletown",
      "New Gribble",
      "San Gribble",
      "Gribble Vista"
   };
   int maxtemps[Years][Cities]=         // двумерный массив
   {
      {96, 100, 87, 101, 105},          // значения для maxtemps[0]
      {96, 98, 91, 107, 104},           // значения для maxtemps[1]
      {97, 101, 93, 108, 107},          // значения для maxtemps[2]
      {98, 103, 95, 109, 108}           // значения для maxtemps[3]
   };
   cout << "Maximum temperatures for 2008 - 2011\n\n";
         // Максимальные температуры в 2008-2011 гг.
   for (int city = 0; city < Cities; ++city)
   {
      cout << cities[city] << ":\t";
      for (int year = 0; year < Years; ++year)
         cout << maxtemps[year][city] << "\t";
      cout << endl;
   }
   // cin.get();
   return 0;
}
```

Ниже показан вывод программы из листинга 5.20:

```
Maximum temperatures for 2008 — 2011

Gribble City:     96       96       97       98
Gribbletown:      100      98       101      103
New Gribble:      87       91       93       95
San Gribble:      101      107      108      109
Gribble Vista:    105      104      107      108
```

Применение знаков табуляции позволяет разместить данные более равномерно, чем с помощью пробелов. Однако разные установки позиций табуляции могут привести к тому, что в различных системах вывод будет выглядеть немного по-разному. В главе 17 будут представлены более точные, но и более сложные методы форматирования вывода.

В грубом варианте можно было бы использовать массив массивов `char` вместо массива указателей для строковых данных. Объявление выглядело бы следующим образом:

```
char cities[Cities][25] =          // массив из 5 массивов по 25 символов
{
   "Gribble City",
   "Gribbletown",
   "New Gribble",
   "San Gribble",
   "Gribble Vista"
};
```

При таком подходе длина каждой из 5 строк ограничивается максимум 24 символами. Массив указателей сохраняет адреса пяти строковых литералов, но массив массивов `char` копирует каждый из пяти строковых литералов в соответствующий массив из 25 символов. Это значит, что массив указателей более экономичен в отношении используемой памяти. Однако если вы намерены модифицировать любую из этих пяти строк, то в этом случае двумерный массив символов будет более удачным вариантом. Это довольно странно, но оба варианта используют одинаковый список инициализации и один и тот же код циклов `for` для отображения строк.

К тому же вы можете использовать массив объектов класса `string` вместо массива указателей для сохранения данных. Объявление будет выглядеть следующим образом:

```
const string cities[Cities] =      // массив из 5 строк
{
   "Gribble City",
   "Gribbletown",
   "New Gribble",
   "San Gribble",
   "Gribble Vista"
};
```

Если вам нужны модифицируемые строки, можете опустить квалификатор `const`. Эта форма будет использовать тот же список инициализации и тот же цикл `for` для отображения строк, как и две другие формы. Если строки будут модифицируемыми, то свойство автоматического изменения размера класса `string` делает такой подход более удобным, чем применение двумерного массива символов.

Резюме

В C++ представлены три варианта циклов: `for`, `while` и `do while`. Цикл позволяет повторно выполнять один и тот же набор инструкций до тех пор, пока проверочное условие цикла оценивается как `true` или не ноль, и цикл прекращает их выполнение, когда это проверочное условие возвращает `false` или 0. Циклы `for` и `while` являются циклами с проверкой на входе, это означает, что они оценивают проверочное условие перед выполнением операторов, находящихся в теле цикла. Цикл `do while` проверяет условие на выходе, т.е. после выполнения операторов, содержащихся в его теле.

Синтаксис каждого цикла позволяет размещать в теле только один оператор. Однако этот оператор может быть составным, или блоком, в виде группы операторов, заключенных в фигурные скобки.

Сравнивающие выражения (выражения отношений), которые сравнивают два значения, часто применяются в качестве проверочных условий цикла. Эти выражения формируются с использованием одной из шести операций отношений: <, <=, ==, >=, > и !=. Сравнивающие выражения возвращают значения типа `bool`: `true` или `false`.

Многие программы читают текстовый ввод или текстовые файлы символ за символом. Класс `istream` предлагает несколько способов делать это. Если `ch` — переменная типа `char`, то следующий оператор читает очередной символ ввода в `ch`:

```
cin >> ch;
```

Однако при этом пропускаются пробелы, символы новой строки и символы табуляции. Показанный ниже вызов функции-члена читает очередной входной символ, независимо от его значения, и помещает его в `ch`:

```
cin.get(ch);
```

Вызов функции-члена `cin.get()` возвращает следующий символ ввода, включая пробелы, символы новой строки и символы табуляции, поэтому он может применяться следующим образом:

```
ch = cin.get();
```

Функция-член `cin.get(char)` сообщает о встреченном состоянии EOF, возвращая значение, которое преобразуется в тип `bool` как `false`, в то время как функция-член `cin.get()` сообщает о EOF, возвращая значение `EOF`, определенное в заголовочном файле `iostream`.

Вложенный цикл — это цикл, находящийся в теле другого цикла `for`. Вложенные циклы обеспечивают естественный способ обработки двумерных массивов.

Вопросы для самоконтроля

1. В чем состоит разница между циклами с проверкой на входе и циклами с проверкой на выходе? Какой из циклов C++ к какой категории относится?
2. Что напечатает следующий фрагмент кода, если использовать его в программе?

```
int i;
for (i = 0; i < 5; i++)
   cout << i;
   cout << endl;
```

3. Что напечатает следующий фрагмент кода, если использовать его в программе?

```
int j;
for (j = 0; j < 11; j += 3)
   cout << j;
cout << endl << j << endl;
```

4. Что напечатает следующий фрагмент кода, если использовать его в программе?

```
int j = 5;
while ( ++j < 9)
   cout << j++ << endl;
```

5. Что напечатает следующий фрагмент кода, если использовать его в программе?

```
int k = 8;
do
   cout << " k = " << k << endl;
while (k++ < 5);
```

6. Напишите цикл `for`, который печатает значения 1 2 4 8 16 32 64, увеличивая вдвое значение переменной счетчика на каждом шаге.
7. Как сделать так, чтобы тело цикла включало более одного оператора?
8. Правилен ли следующий оператор? Если нет, то почему? Если да, то что он делает?

```
int x = (1,024);
```

А правилен ли такой оператор?

```
int y;
y = 1,024;
```

9. Чем отличается `cin>>ch` от `cin.get(ch)` и `ch=cin.get()` с точки зрения ввода?

Упражнения по программированию

1. Напишите программу, запрашивающую у пользователя ввод двух целых чисел. Затем программа должна вычислить и выдать сумму всех целых чисел, лежащих между этими двумя целыми. Предполагается, что меньшее значение вводится первым. Например, если пользователь ввел 2 и 9, программа должна сообщить, что сумма всех целых чисел от 2 до 9 равна 44.
2. Перепишите код из листинга 5.4 с использованием объекта `array` вместо встроенного массива и типа `long double` вместо `long long`. Найдите значение 100!
3. Напишите программу, которая приглашает пользователя вводить числа. После каждого введенного значения программа должна выдавать накопленную сумму введенных значений. Программа должна завершаться при вводе `0`.
4. Дафна инвестировала $100 под простые 10%. Другими словами, ежегодно инвестиция должна приносить 10% инвестированной суммы, т.е. $10 каждый год:

   прибыль = 0,10 × исходный баланс

   В то же время Клео инвестировала $100 под сложные 5%. Это значит, что прибыль составит 5% от текущего баланса, включая предыдущую накопленную прибыль:

   прибыль = 0,05 × текущий баланс

   Клео зарабатывает 5% от $100 в первый год, что дает ей $105. На следующий год она зарабатывает 5% от $105, что составляет $5.25, и т.д. Напишите программу, которая вычислит, сколько лет понадобится для того, чтобы сумма баланса Клео превысила сумму баланса Дафны, с отображением значений обоих балансов за каждый год.
5. Предположим, что вы продаете книгу по программированию на языке C++ для начинающих. Напишите программу, которая позволит ввести ежемесячные объемы продаж в течение года (в количестве книг, а не в деньгах). Программа должна использовать цикл, в котором выводится приглашение с названием месяца, применяя массив указателей на `char` (или массив объектов `string`, если вы предпочитаете его), инициализированный строками — названиями месяцев, и сохраняя введенные значения в массиве `int`. Затем программа должна найти сумму содержимого массива и выдать общий объем продаж за год.

6. Выполните упражнение 5, но используя двумерный массив для сохранения данных о месячных продажах за 3 года. Выдайте общую сумму продаж за каждый год и за все годы вместе.

7. Разработайте структуру по имени `car`, которая будет хранить следующую информацию об автомобиле: название производителя в виде строки в символьном массиве или в объекте `string`, а также год выпуска автомобиля в виде целого числа. Напишите программу, которая запросит пользователя, сколько автомобилей необходимо включить в каталог. Затем программа должна применить `new` для создания динамического массива структур `car` указанного пользователем размера. Далее она должна пригласить пользователя ввести название производителя и год выпуска для наполнения данными каждой структуры в массиве (см. главу 4). И, наконец, она должна отобразить содержимое каждой структуры. Пример запуска программы должен выглядеть подобно следующему:

```
Сколько автомобилей поместить в каталог? 2
Автомобиль #1:
Введите производителя: Hudson Hornet
Укажите год выпуска: 1952
Автомобиль #2:
Введите производителя: Kaiser
Укажите год выпуска: 1951
Вот ваша коллекция:
1952 Hudson Hornet
1951 Kaiser
```

8. Напишите программу, которая использует массив `char` и цикл для чтения по одному слову за раз до тех пор, пока не будет введено слово `done`. Затем программа должна сообщить количество введенных слов (исключая `done`). Пример запуска должен быть таким:

```
Вводите слова (для завершения введите слово done):
anteater birthday category dumpster
envy finagle geometry done for sure
Вы ввели 7 слов.
```

Вы должны включить заголовочный файл `cstring` и применять функцию `strcmp()` для выполнения проверки.

9. Напишите программу, соответствующую описанию программы из упражнения 8, но с использованием объекта `string` вместо символьного массива. Включите заголовочный файл `string` и применяйте операции отношений для выполнения проверки.

10. Напишите программу, использующую вложенные циклы, которая запрашивает у пользователя значение количества строк для отображения. Затем она должна отобразить указанное число строк со звездочками, с одной звездочкой в первой строке, двумя — во второй и т.д. В каждой строке звездочкам должны предшествовать точки — в таком количестве, чтобы общее число символов в каждой строке было равно количеству строк. Пример запуска программы должен выглядеть следующим образом:

```
Введите количество строк: 5
....*
...**
..***
.****
*****
```

6

Операторы ветвления и логические операции

В ЭТОЙ ГЛАВЕ...

- Оператор `if`
- Оператор `if else`
- Логические операции: `&&`, `||` и `!`
- Библиотека символьных функций `cctype`
- Условная операция `?:`
- Оператор `switch`
- Операторы `continue` и `break`
- Циклы чтения чисел
- Базовый файловый ввод-вывод

Одним из ключевых аспектов проектирования интеллектуальных программ является предоставление им возможности принятия решений. В главе 5 был продемонстрирован один из видов принятия решений – цикл, при котором программа решает, следует ли продолжать циклическое выполнение части кода. Здесь же мы исследуем, как C++ позволяет с помощью операторов ветвления принимать решения относительно выполнения одного из альтернативных действий. Какое средство защиты от вампиров (чеснок или крест) следует избрать? Какой пункт меню выбрал пользователь? Ввел ли пользователь ноль? Для принятия подобных решений в C++ предусмотрены операторы `if` и `switch`, и именно они будут предметом рассмотрения настоящей главы. Здесь мы также поговорим об условной операции, которая предоставляет другой способ принятия решений, а также логических операциях, позволяющих комбинировать две проверки в одну. И, наконец, в этой главе вы также впервые ознакомитесь с файловым вводом-выводом.

Оператор `if`

Когда программа C++ должна принять решение о том, какое из альтернативных действий следует выполнить, такой выбор обычно реализуется оператором `if`. Этот оператор имеет две формы: просто `if` и `if else`. Давайте сначала исследуем простой `if`. Он создан по образцу обычного английского языка, как в выражении "If you have a Captain Cookie card, you get a free cookie" (игра слов на основе созвучности фамилии Кук и слова "cookie" (печенье) – *прим. перев.*). Оператор `if` разрешает программе выполнять оператор или блок операторов при условии истинности проверочного условия, и пропускает этот оператор или блок, если проверочное условие оценивается как ложное. Таким образом, оператор `if` позволяет программе принимать решение относительно того, нужно ли выполнять некоторую часть кода.

Синтаксис оператора `if` подобен `while`:

```
if (проверочное-условие)
   оператор
```

Истинность выражения *проверочное-условие* заставляет программу выполнить *оператор*, который может быть единственным оператором или блоком операторов. Ложность выражения *проверочное-условие* заставляет программу пропустить *оператор* (рис. 6.1). Как и с проверочными условиями циклов, тип проверочного условия `if` приводится к `bool`, поэтому ноль трактуется как `false`, а все, что отличается от нуля – как `true`. Вся конструкция `if` рассматривается как одиночный оператор.

Чаще всего *проверочное-условие* – выражение сравнения, вроде тех, которые управляют циклами. Например, предположим, что вы хотите запрограммировать подсчет пробелов во входной строке, а также общее количество символов. Для чтения символов можно использовать оператор `cin.get(char)` внутри цикла `while`, а затем с помощью оператора `if` идентифицировать и подсчитывать пробельные символы. В листинге 6.1 реализован такой алгоритм, при этом точка служит признаком конца входного предложения.

Листинг 6.1. `if.cpp`

```
// if.cpp -- использование оператора if
#include <iostream>
int main()
{
   using std::cin;   // объявления using
   using std::cout;
```

```
    char ch;
    int spaces = 0;
    int total = 0;
    cin.get(ch);
    while (ch != '.')           // завершение по окончании предложения
    {
       if (ch == ' ')           // проверка ch на равенство пробелу
          ++spaces;
       ++total;                 // выполняется на каждом шаге цикла
       cin.get(ch);
    }
    cout << spaces << " spaces, " << total;    // вывод количества пробелов
                                               // и символов в предложении
    cout << " characters total in sentence\n";
    return 0;
}
```

```
оператор1
if (проверочное_выражение)
      оператор2
оператор3
```

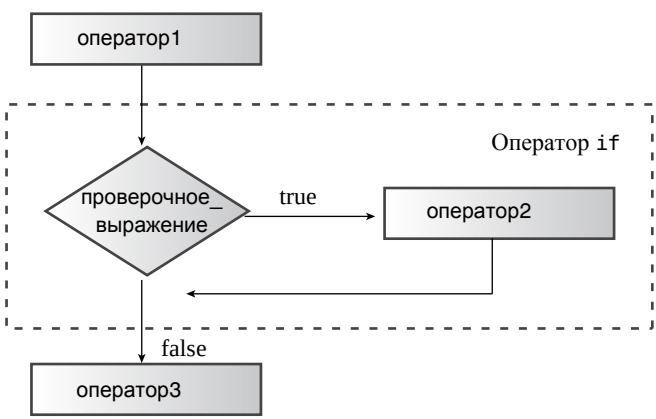


***Рис. 6.1.** Структура оператора* `if`

Ниже показан пример вывода программы из листинга 6.1:

```
The balloonist was an airhead
with lofty goals.
6 spaces, 46 characters total in sentence
```

Как следует из комментариев в листинге 6.1, оператор `++spaces;` выполняется только в том случае, если `ch` равен пробелу. Поскольку оператор `++total;` находится вне `if`, он выполняется на каждом шаге цикла. Обратите внимание, что в общее количество символов входит также символ новой строки, который генерируется нажатием клавиши <Enter>.

Оператор `if else`

В то время как оператор `if` позволяет программе принять решение о том, должен ли выполняться *определенный* оператор или блок, `if else` позволяет решить, какой из двух операторов или блоков следует выполнить. Это незаменимое средство для

программирования альтернативных действий. Оператор C++ `if else` моделирует простой английский язык, как в предложении "If you have a Captain Cookie card, you get a Cookie Plus Plus, else you just get a Cookie d'Ordinaire" (непереводимая игра слов с применением местных идиоматических выражений — *прим. перев.*).

Оператор `if else` имеет следующую общую форму:

```
if (проверочное-условие)
   оператор1
else
   оператор2
```

Если *проверочное-условие* равно `true` или не ноль, то программа выполняет *оператор1* и пропускает *оператор2*. В противном случае, когда *проверочное-условие* равно `false` или ноль, программа выполняет *оператор2* и пропускает *оператор1*. Потому следующий фрагмент кода печатает первое сообщение, если `answer` равно `1492`, и второе — в противном случае:

```
if (answer == 1492)
   cout << "That's right!\n";
else
   cout << "You'd better review Chapter 1 again.\n";
```

Каждый оператор может быть либо отдельным оператором, либо блоком операторов, заключенным в фигурные скобки (рис. 6.2). Вся конструкция `if else` трактуется синтаксически как одиночный оператор.

Например, предположим, что вы хотите преобразовать входящий текст, шифруя буквы и оставляя нетронутыми символы новой строки. Это значит, что нужно заставить программу выполнять одно действие для символов новой строки и другое — для всех прочих символов. Как показано в листинге 6.2, оператор `if else` позволяет легко решить эту задачу. В данном листинге также иллюстрируется применение квалификатора `std::` — одной из альтернатив директивы `using`.

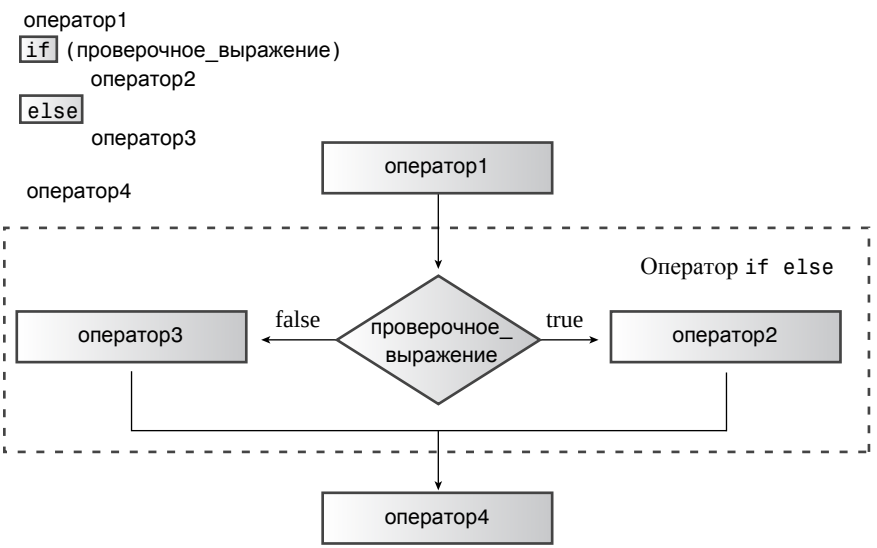


Рис. 6.2. Структура оператора `if else`

Листинг 6.2. ifelse.cpp

```
// ifelse.cpp -- использование оператора if else
#include <iostream>
int main()
{
   char ch;
   std::cout << "Type, and I shall repeat.\n"; // запрос на ввод строки
   std::cin.get(ch);

   while (ch != '.')
   {
      if (ch == '\n')
         std::cout << ch;          // выполнение в случае символа новой строки
      else
         std::cout << ++ch;        // выполнение в противном случае
      std::cin.get(ch);
   }

   // попробуйте ch + 1 вместо ++ch, чтобы увидеть интересный эффект
   std::cout << "\nPlease excuse the slight confusion.\n";

   // std::cin.get();
   // std::cin.get();
   return 0;
}
```

Ниже показан пример вывода программы из листинга 6.2:

```
Type, and I shall repeat.
An ineffable joy suffused me as I beheld
Bo!jofggbcmf!kpz!tvggvtfe!nf!bt!J!cfifme
the wonders of modern computing.
uif!xpoefst!pg!npefso!dpnqvujoh
Please excuse the slight confusion.
```

Обратите внимание, что в одном из комментариев в листинге 6.2 предлагается заменить ++ch на ch + 1, чтобы увидеть интересный эффект. Можете ли вы предположить, что произойдет? Если нет, сделайте это и посмотрите, что получится, после чего попробуйте объяснить. (Подсказка: это касается того, как cout обрабатывает разные типы данных.)

Форматирование операторов if else

Имейте в виду, что две альтернативы в операторе if else должны быть одиночными операторами. Если в каждой логической ветви требуется более одного оператора, воспользуйтесь фигурными скобками, чтобы организовать их в единый блок. В отличие от других языков, таких как BASIC и FORTRAN, C++ не воспринимает автоматически все, что находится между if и else, как один блок, поэтому необходимо с помощью фигурных скобок объединять операторы в блок. Следующий код, например, вызовет ошибку во время компиляции:

```
if (ch == 'Z')
   zorro++;          // if заканчивается здесь
   cout << "Another Zorro candidate\n";
else                 // неверно
   dull++;
cout << "Not a Zorro candidate\n";
```

Компилятор рассматривает это как простой оператор if, который заканчивается на zorro++;. Затем идет оператор cout. До этого места все хорошо. Но далее идет то, что воспринимается компилятором как бесхозный else, а потому он считает это синтаксической ошибкой.

Чтобы код делал то, что нужно, следует указать фигурные скобки:

```
if (ch == 'Z')
{                       // блок, выполняемый, если условие истинно
   zorro++;
   cout << "Another Zorro candidate\n";
}
else
{                       // блок, выполняемый, если условие ложно
   dull++;
   cout << "Not a Zorro candidate\n";
}
```

Поскольку C++ — язык свободной формы, фигурные скобки можно размещать как вам угодно, до тех пор, пока они ограничивают операторы языка. В предыдущем примере демонстрируется один популярный формат. А вот и другой формат:

```
if (ch == 'Z') {
   zorro++;
   cout << "Another Zorro candidate\n";
   }
else {
   dull++;
   cout << "Not a Zorro candidate\n";
   }
```

Первая форма подчеркивает блочную структуру операторов, в то время как вторая более тесно связывает блоки с ключевыми словами if и else. Любой стиль ясен и согласован, а потому будет служить вам хорошо; однако вы можете столкнуться с руководителем или работодателем, который имеет собственные строгие и специфические взгляды на эту тему.

Конструкция if else if else

В компьютерных программах, как и в жизни, иногда приходится выбирать из более чем двух вариантов. Оператор C++ if else можно расширить, чтобы он отвечало таким потребностям. Как уже говорилось, за else должен следовать единственный оператор, который может быть и блоком операторов. Поскольку конструкция if else сама является единым оператором, она может следовать за else:

```
if (ch == 'A')
   a_grade++;          // альтернатива # 1
else
   if (ch == 'B')      // альтернатива # 2
      b_grade++;       // подальтернатива # 2a
   else
      soso++;          // подальтернатива # 2b
```

Если значение ch не равно 'A', программа переходит к else. Там второй оператор if else разделяет эту альтернативу еще на два варианта. Свойство свободного форматирования C++ позволяет расположить эти элементы в более читабельном виде:

```
if (ch == 'A')
   a_grade++;        // альтернатива # 1
else if (ch == 'B')
   b_grade++;        // альтернатива # 2
else
   soso++;           // альтернатива # 3
```

Это выглядит как совершенно новая управляющая структура — `if else if else`. Но на самом деле это один оператор `if else`, вложенный в другой. Пересмотренный формат выглядит намного яснее и позволяет даже при поверхностном взгляде оценить все альтернативы. Вся эта конструкция по-прежнему трактуется как единственный оператор.

В листинге 6.3 это форматирование используется для построения простой программы загадок и отгадок.

Листинг 6.3. `ifelseif.cpp`

```
// ifelseif.cpp -- использование оператора if else if else
#include <iostream>
const int Fave = 27;
int main()
{
   using namespace std;
   int n;
   cout << "Enter a number in the range 1-100 to find ";
   cout << "my favorite number: "; // запрос на ввод числа из диапазона 1-100
   do
   {
      cin >> n;
      if (n < Fave)
         cout << "Too low -- guess again: ";      // число слишком мало
      else if (n > Fave)
         cout << "Too high -- guess again: ";     // число слишком велико
      else
         cout << Fave << " is right!\n";          // число угадано
   } while (n != Fave);
   return 0;
}
```

Ниже представлен пример вывода программы из листинга 6.3:

```
Enter a number in the range 1-100 to find my favorite number: 50
Too high -- guess again: 25
Too low -- guess again: 37
Too high -- guess again: 31
Too high -- guess again: 28
Too high -- guess again: 27
27 is right!
```

Условные операции и предотвращение ошибок

Многие программисты превращают более интуитивно понятное выражение *переменная == значение* в *значение == переменная*, чтобы предотвратить ошибки, связанные с опечатками, когда вместо операции проверки равенства (==) вводится операция присваивания (=). Например, следующее условие корректно и будет работать правильно:

```
if (3 == myNumber)
```

Однако если допустить ошибку и ввести оператор, как показано ниже, то компилятор выдаст сообщение об ошибке, поскольку расценит это как попытку присвоить значение литералу (3 всегда равно 3, и ему нельзя присвоить ничего другого):

```
if (3 = myNumber)
```

Предположим, что сделана та же опечатка, но используется обычный формат:

```
if (myNumber = 3)
```

В этом случае компилятор просто присвоит значение 3 переменной myNumber, и блок внутри if будет выполнен — очень распространенная ошибка, которую трудно обнаружить. (Однако многие компиляторы будут выдавать предупреждение, на которое разумно обратить внимание.) В качестве общего правила запомните следующее: написать код, позволяющий компилятору обнаружить ошибку, гораздо легче, чем разбираться с непонятными мистическими результатами неверного выполнения программ.

Логические выражения

Часто приходится проверять более одного условия. Например, чтобы символ относился к прописным буквам, его значение должно быть больше или равно 'a' и меньше или равно 'z'. Либо же, если вы просите пользователя ответить y или n, то наряду с прописными должны приниматься и заглавные буквы (Y или N). Чтобы обеспечить такие возможности, C++ предлагает три логических операции, с помощью которых можно комбинировать или модифицировать существующие выражения: логическое "ИЛИ" (которое записывается как ||), логическое "И" (записывается как &&) и логическое "НЕ" (записывается как !). Рассмотрим каждую из них.

Логическая операция "ИЛИ": ||

В английском языке слово "or" (или) означает, что одно из двух условий либо оба сразу удовлетворяют некоторому требованию. Например, вы можете поехать на пикник, устроенный компанией MegaMicro, если вы *или* ваш(а) супруг(а) работаете в этой компании. Эквивалент логической операции "ИЛИ" в C++ записывается как ||. Эта операция комбинирует два выражения в одно. Если одно или оба исходных выражения возвращают true или не ноль, то результирующее выражение имеет значение true (истина). В противном случае выражение имеет значение false. Ниже показаны некоторые примеры:

```
5 == 5 || 5 == 9      // истинно, потому что первое выражение истинно
5 > 3 || 5 > 10       // истинно, потому что первое выражение истинно
5 > 8 || 5 < 10       // истинно, потому что второе выражение истинно
5 < 8 || 5 > 2        // истинно, потому что оба выражения истинны
5 > 8 || 5 < 2        // ложно, потому что оба выражения ложны
```

Поскольку || имеет более низкий приоритет, чем операции сравнения, нет необходимости использовать в этих выражениях скобки. В табл. 6.1 показано, как работает операция ||.

Таблица 6.1. Операция ||

Значение expr1 || expr2		
	expr1 == true	expr1 == false
expr2 == true	true	true
expr2 == false	true	false

В C++ предполагается, что операция || является *точкой следования*. Это значит, что любое изменение, проведенное в левой части, вычисляется прежде, чем вычисляется правая часть. (Или, как сейчас принято в C++11, подвыражение слева от операции находится с точки зрения последовательности перед подвыражением справа от операции.) Например, рассмотрим следующее выражение:

```
i++ < 6 || i == j
```

Предположим, что изначально переменная i имела значение 10. К моменту сравнения с j переменная i получает значение 11. Таким образом, C++ не заботится о правой части выражения, если выражение слева истинно, потому что одного истинного выражения достаточно, чтобы все составное выражение было оценено как истинное. (Вспомните, что операции точки с запятой и запятой также являются точками следования.)

В листинге 6.4 используется операцию || внутри if для того, чтобы проверить заглавную и прописную версии символа. К тому же применяется средство конкатенации строк C++ (см. главу 4) для разнесения одной строки на три строки в коде.

Листинг 6.4. or.cpp

```
// or.cpp -- использование логической операции "ИЛИ"
#include <iostream>
int main()
{
   using namespace std;
   cout << "This program may reformat your hard disk\n"
           "and destroy all your data.\n"
           "Do you wish to continue? <y/n> ";
   char ch;
   cin >> ch;
   if (ch == 'y' || ch == 'Y')                // y или Y
      cout << "You were warned!\a\a\n";
   else if (ch == 'n' || ch == 'N')           // n или N
      cout << "A wise choice ... bye\n";
   else
      cout << "That wasn't a y or n! Apparently you "
              "can't follow\ninstructions, so "
              "I'll trash your disk anyway.\a\a\a\n";
   return 0;
}
```

Вот как выглядит пример выполнения программы из листинга 6.4:

```
This program may reformat your hard disk
and destroy all your data.
Do you wish to continue? <y/n> N
A wise choice ... bye
```

Эта программа читает только один символ, поэтому принимается во внимание только первый символ ответа. Это значит, что пользователь может ввести **NO!** вместо **N**, но программа прочитает только **N**. Но если ей пришлось бы читать еще входные символы, то первым оказался бы символ **O**.

Логическая операция "И": &&

Логическая операция "И", которая записывается как &&, также комбинирует два выражения в одно. Результирующее выражение имеет значение true только в том случае, когда оба исходных выражения также равны true.

Ниже приведены некоторые примеры:

```
5 == 5 && 4 == 4      // истинно, потому что оба выражения истинны
5 == 3 && 4 == 4      // ложно, потому что первое выражение ложно
5 > 3 && 5 > 10       // ложно, потому что второе выражение ложно
5 > 8 && 5 < 10       // ложно, потому что первое выражение ложно
5 < 8 && 5 > 2        // истинно, потому что оба выражения истинны
5 > 8 && 5 < 2        // ложно, потому что оба выражения ложны
```

Поскольку `&&` имеет меньший приоритет, чем операции сравнения, нет необходимости использовать скобки. Подобно `||`, операция `&&` действует как точка следования, а потому возможны любые побочные эффекты перед тем, как будет вычислено правое выражение. Если левое выражение ложно, то и все составное выражение также ложно, поэтому в таком случае C++ можно не беспокоиться о вычислении правой части. Работа операции `&&` описана в табл. 6.2.

Таблица 6.2. Операция `&&`

	Значение `expr1 && expr2`	
	`expr1 == true`	`expr1 == false`
`expr2 == true`	`true`	`false`
`expr2 == false`	`false`	`false`

В листинге 6.5 демонстрируется использование `&&` в стандартной ситуации, когда нужно прервать цикл `while` по двум разным причинам. В этом листинге цикл `while` читает значения в массив. Одно условие (`i < ArSize`) прерывает цикл, когда массив заполнен. Вторая (`temp >= 0`) предоставляет пользователю возможность прервать цикл раньше, введя отрицательное значение. Программа использует операцию `&&`, чтобы скомбинировать эти две проверки в одно условие. В программе также присутствуют два оператора `if`, один оператор `if else` и цикл `for`, т.е. иллюстрируется несколько тем из этой главы и главы 5.

Листинг 6.5. `and.cpp`

```
// and.cpp -- использование логической операции "И"
#include <iostream>
const int ArSize = 6;
int main()
{
   using namespace std;
   float naaq[ArSize];
   cout << "Enter the NAAQs (New Age Awareness Quotients) "
        << "of\nyour neighbors. Program terminates "
        << "when you make\n" << ArSize << " entries "
        << "or enter a negative value.\n";
   int i = 0;
   float temp;
   cout << "First value: ";          // ввод первого значения
   cin >> temp;
   while (i < ArSize && temp >= 0)  // два критерия завершения
   {
      naaq[i] = temp;
      ++i;
      if (i < ArSize)               // в массиве еще есть место
      {
         cout << "Next value: ";
         cin >> temp;               // ввод следующего значения
      }
   }
```

```
   if (i == 0)
      cout << "No data--bye\n";    // данные отсутствуют
   else
   {
      cout << "Enter your NAAQ: ";
      float you;
      cin >> you;
      int count = 0;

      for (int j = 0; j < i; j++)
         if (naaq[j] > you)
            ++count;

      cout << count;
      cout << " of your neighbors have greater awareness of\n"
           << "the New Age than you do.\n";
   }
   return 0;
}
```

Обратите внимание, что программа в листинге 6.5 помещает вывод во временную переменную `temp`. И только после того, как введенное значение будет проверено, и станет ясно, что введено корректное значение, программа помещает новое значение в массив.

Ниже показаны два примера выполнения программы. В первом примере программа прерывается после шести введенных значений:

```
Enter the NAAQs (New Age Awareness Quotients) of
your neighbors. Program terminates when you make
6 entries or enter a negative value.
First value: 28
Next value: 72
Next value: 15
Next value: 6
Next value: 130
Next value: 145
Enter your NAAQ: 50
3 of your neighbors have greater awareness of
the New Age than you do.
```

Второй раз программа прерывается после ввода отрицательного значения:

```
Enter the NAAQs (New Age Awareness Quotients) of
your neighbors. Program terminates when you make
6 entries or enter a negative value.
First value: 123
Next value: 119
Next value: 4
Next value: 89
Next value: -1
Enter your NAAQ: 123.031
0 of your neighbors have greater awareness of
the New Age than you do.
```

Замечания по программе

Ниже представлена часть программы из листинга 6.5, отвечающая за ввод:

```
cin >> temp;
while (i < ArSize && temp >= 0)     // два критерия завершения
{
   naaq[i] = temp;
   ++i;
   if (i < ArSize)                  // в массиве еще есть место
   {
      cout << "Next value: ";
      cin >> temp;                  // ввод следующего значения
   }
}
```

Программа начинается с чтения первого входного значения во временную переменную по имени `temp`. Затем с помощью проверочного условия цикла `while` выясняется, есть ли еще место в массиве (`i < ArSize`), и не является ли введенное значение отрицательным (`temp >= 0`). Если это так, программа копирует значение `temp` в массив и увеличивает индекс массива на единицу. В этот момент, поскольку нумерация массива начинается с нуля, `i` равно общему количеству введенных значений. Значит, если `i` начинается с нуля, то на первом проходе цикла осуществляется присваивание значения `naaq[0]` и установка `i` в 1.

Цикл прерывается, когда массив будет заполнен, либо когда пользователь введет отрицательное значение. Обратите внимание, что цикл читает новое значение в `temp`, только если `i` меньше, чем `ArSize` — т.е. только если в массиве еще есть место.

После получения данных программа использует оператор `if else` для проверки того, вводились ли вообще данные (когда первым же введенным элементом было отрицательное число), и обрабатывает данные, если они есть.

Установка диапазонов с помощью &&

Операция `&&` также позволяет установить последовательность операторов `if else if else`, где каждый выбор соответствует определенному диапазону значений. В листинге 6.6 иллюстрируется такой подход. В нем также демонстрируется полезная техника обработки серии сообщений. Точно так же, как переменная-указатель на `char` может идентифицировать целую строку, указывая на ее начало, массив указателей на `char` может идентифицировать серию строк. Вы просто присваиваете адрес каждой строки различным элементам массива. Код в листинге 6.6 использует массив `qualify` для хранения адресов четырех строк. Например, `qualify[1]` содержит адрес строки `"mud tug-of-war\n"`. Программа затем может использовать `qualify[1]` как любой другой указатель на строку — например, вместе с `cout` либо при вызове `strlen()` или `strcmp()`. Применение квалификатора `const` защищает эти строки от непреднамеренных изменений.

Листинг 6.6. more_and.cpp

```
// more_and.cpp -- использование логической операции "И"
#include <iostream>
const char * qualify[4] =           // массив указателей на строки
{
   "10,000-meter race.\n",          // забег на 10 000 метров
   "mud tug-of-war.\n",             // перетягивание каната в грязи
   "masters canoe jousting.\n",     // состязания мастеров каноэ
   "pie-throwing festival.\n"       // фестиваль по бросанию пирожков
};
int main()
{
```

```
   using namespace std;
   int age;
   cout << "Enter your age in years: ";           // запрос возраста в годах
   cin >> age;
   int index;
   if (age > 17 && age < 35)
      index = 0;
   else if (age >= 35 && age < 50)
      index = 1;
   else if (age >= 50 && age < 65)
      index = 2;
   else
      index = 3;
   cout << "You qualify for the " << qualify[index]; // вывод рекомендованного результата
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 6.6:

```
Enter your age in years: 87
You qualify for the pie-throwing festival.
```

Введенный возраст не соответствует ни одному из проверяемых диапазонов, поэтому программа присваивает `index` значение 3 и затем печатает соответствующую строку.

Замечания по программе

В листинге 6.6 выражение `age > 17 && age < 35` проверяет возраст на предмет попадания в диапазон между двумя значениями, т.е. он должен быть от 18 до 34 лет включительно. Выражение `age >= 35 && age < 50` использует операцию >= для включения в диапазон 35, т.е. представляет диапазон возрастов от 35 до 49 включительно. Если бы в программе применялось выражение `age > 35 && age < 50`, то значение 35 было бы потеряно для всех проверок. Организуя проверки диапазонов, вы должны следить, чтобы диапазоны не имели промежутков между собой, а также не перекрывались. К тому же следует удостовериться в том, что диапазоны заданы корректно; см. врезку "Проверка диапазонов" ниже.

Оператор `if else` служит для выбора индекса массива, который, в свою очередь, идентифицирует определенную строку.

Проверка диапазонов

Обратите внимание, что каждая часть проверки диапазонов должна использовать операцию "И" для объединения двух полных сравнительных выражений:

```
if (age > 17 && age < 35)   // Нормально
```

Не заимствуйте из математики и не применяйте следующую нотацию:

```
if (17 < age < 35)          // Не делайте так!
```

Если вы допустите ошибку подобного рода, компилятор не сможет ее обнаружить, т.к. это корректный синтаксис C++. Операция ассоциируется слева направо, поэтому последнее выражение эквивалентно такому:

```
if ((17 < age) < 35)
```

Но `17 < age` может быть либо `true` (или 1), либо `false` (или 0). Это значит, что в любом случае выражение `17 < age` меньше 35, а потому выражение `(17 < age) < 35` в результате всегда будет давать `true`!

Логическая операция "НЕ": !

Операция ! выполняет отрицание, или обращает, истинность выражения, следующего за ней. То есть если expression равно true, то !expression равно false, и наоборот. Точнее говоря, если expression имеет значение true, или ненулевое, то !expression будет равно false.

Обычно выражение отношения можно представить яснее без применения операции !:

```
if (!(x > 5))   // в этом случае if (x <= 5) яснее
```

Однако операция ! может быть полезна с функциями, которые возвращают значения true/false либо значения, которые могут интерпретироваться подобным образом. Например, strcmp(s1, s2) возвращает не ноль (true), если две строки в стиле C, s1 и s2, отличаются друг от друга, и ноль, если они одинаковы. Это значит, что !strcmp(s1, s2) равно true, если две строки эквивалентны.

В листинге 6.7 используется прием применения операции ! к значению, которое возвращается функцией проверки числового ввода на предмет возможности его присваивания типу int. Пользовательская функция is_int(), которая будет рассматриваться позже, возвращает true, если ее аргумент находится в диапазоне допустимых значений для присваивания типу int. Затем программа применяет проверку условия while(!is_int(num)), чтобы отклонить значения, которые не входят в диапазон.

Листинг 6.7. not.cpp

```
// not.cpp -- использование логической операции "НЕ"
#include <iostream>
#include <climits>
bool is_int(double);
int main()
{
   using namespace std;
   double num;
   cout << "Yo, dude! Enter an integer value: "; // запрос на ввод целочисленного значения
   cin >> num;
   while (!is_int(num))                  // продолжать, пока num не является int
   {
      cout << "Out of range -- please try again: "; // выход за пределы диапазона
      cin >> num;
   }
   int val = int (num);                  // приведение типа
   cout << "You've entered the integer " << val << "\nBye\n";
   return 0;
}
bool is_int(double x)
{
   if (x <= INT_MAX && x >= INT_MIN)     // проверка предельных значений climits
      return true;
   else
      return false;
}
```

Ниже показан пример выполнения программы из листинга 6.7 в системе с 32-битовым типом int:

```
Yo, dude! Enter an integer value: 6234128679
Out of range -- please try again: -8000222333
Out of range -- please try again: 99999
You've entered the integer 99999
Bye
```

Замечания по программе

В случае ввода слишком большого значения при выполнении программы, читающей тип `int`, многие реализации C++ просто усекают значение, не сообщая о потере данных. Программа в листинге 6.7 избегает этого за счет того, что читает потенциальное значение `int` как `double`. Тип `double` имеет более чем достаточную точность для того, чтобы сохранить обычное значение `int`, а его диапазон допустимых значений намного больше. Другим вариантом могло быть сохранение веденного значения в переменной типа `long long`, предполагая, что этот тип шире, чем `int`.

Булевская функция `is_int()` использует две символические константы (`INT_MAX` и `INT_MIN`), определенные в файле `climits` (который обсуждался в главе 3), для проверки, что значение ее аргумента находится в допустимых пределах. Если это так, функция возвращает `true`; в противном случае — `false`.

В функции `main()` используется условие цикла для отклонения неправильного ввода пользователя. Можно сделать программу более дружественной за счет отображения допустимых границ `int`, когда введено неправильное значение. После проверки достоверности введенного значения программа присваивает его переменной типа `int`.

Факты, связанные с логическими операциями

Как упоминалось ранее в этой главе, логические операции "ИЛИ" и "И" в C++ обладают более низким приоритетом, чем операции сравнения. Это значит, что такое выражение, как

```
x > 5 && x < 10
```

интерпретируется следующим образом:

```
(x > 5) && (x < 10)
```

С другой стороны, операция "НЕ" (`!`) имеет более высокий приоритет, чем любая арифметическая операция и операция сравнения. Таким образом, для отрицания выражения его необходимо заключить в скобки:

```
!(x > 5)    // равно false, если x больше 5
!x > 5      // равно true, если !x больше 5
```

Кстати, результатом второго из приведенных выражений всегда будет `false`, т.к. `!x` принимает значения `true` или `false`, что преобразуется, соответственно, в 1 и 0.

Логическая операция "И" имеет более высокий приоритет, чем логическая операция "ИЛИ". Поэтому следующее выражение:

```
age > 30 && age < 45 || weight > 300
```

означает вот что:

```
(age > 30 && age < 45) || weight > 300
```

Здесь первое условие говорит о том, что возраст должен быть от 31 до 44 лет включительно, а второе — что вес должен быть больше 300 (фунтов). Все выражение будет истинным, если истинно одно из выражений или оба сразу.

Разумеется, можно использовать скобки, чтобы явно указать программе, как следует интерпретировать выражение. Например, предположим, что вы хотите использовать && для комбинирования условий о том, что возраст (age) должен быть больше 50 или вес (weight) — больше 300, с условием, что размер пожертвования (donation) должен быть больше 1000. Для этого часть "ИЛИ" потребуется поместить в скобки:

```
(age > 50 || weight > 300) && donation > 1000
```

Иначе компилятор скомбинирует условие weight с условием donation, вместо того чтобы скомбинировать его с условием age.

Хотя правила приоритетов операций C++ часто позволяют писать составные выражения без использования скобок, все же проще всегда применять скобки для группирования проверок, независимо от того, нужны они или нет. Это улучшает читабельность кода, исключает неправильное пониманис порядка приоритетов и снижает вероятность ошибки из-за того, что вы не помните точно правил приоритетов.

C++ гарантирует, что когда программа вычисляет логическое выражение, то делает это слева направо, и прекращает вычисление, как только ответ становится ясен. Предположим, например, что есть следующее выражение:

```
x != 0 && 1.0 / x > 100.0
```

Если первое условие дает false, то и результатом всего выражения будет false. Причина в том, что для возврата true обе части составного выражения должны быть true. Зная, что первое выражение дает false, программе незачем вычислять второе. И это — удача для данного примера, поскольку вычисление второго выражения может привести к делению на ноль, которое не входит в список допустимых действий компьютера.

Альтернативные представления

Не все клавиатуры предоставляют возможность ввода символов, используемых для обозначения логических операций, поэтому в стандарте C++ предусмотрены их альтернативные представления, которые показаны в табл. 6.3. Идентификаторы and, or и not являются зарезервированными словами C++, а это значит, что их нельзя использовать в качестве имен переменных и тому подобного. Они не рассматриваются как ключевые слова, потому что являются альтернативными представлениями существующих средств языка. Кстати, они не являются зарезервированными словами в C, но в программах на C они могут применяться в качестве операций только при условии включения заголовочного файла iso646.h. В C++ этот файл не требуется.

Таблица 6.3. Альтернативные представления логических операций

Операция	Альтернативное представление
&&	and
||	or
!	not

Библиотека символьных функций cctype

Язык C++ унаследовал от C удобный пакет функций для работы с символами, прототипы которых находятся в заголовочном файле cctype (или ctype.h в старом стиле), и это упрощает решение таких задач, как выяснение того, является символ символом верхнего регистра, десятичной цифрой или знаком препинания.

Например, функция `isaplha(ch)` возвращает ненулевое значение, если `ch` — буква, и ноль — в противоположном случае. Аналогично, `ispunct(ch)` возвращает значение `true`, только если `ch` — знак препинания, такой как запятая или точка. (Эти функции возвращают значение типа `int`, а не `bool`, но `int` очень легко преобразуется в `bool`.)

Использовать эти функции намного удобнее, чем операции "И" и "ИЛИ". Например, вот как пришлось бы с помощью "И" и "ИЛИ" проверять, что `ch` является буквенным символом:

```
if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
```

Сравните это с применением функции `isalpha()`:

```
if (isalpha(ch))
```

Однако преимущество заключается не только в том, что `isalpha()` легче использовать. Форма с операциями "И"/"ИЛИ" предполагает, что коды символов, находящихся в пределах A–Z, составляют последовательность, в которую не входят какие-либо другие символы. Это предположение верно для кодировки ASCII, но не всегда правильно в общем случае.

В листинге 6.8 демонстрируется применение некоторых функций из семейства `cctype`. В частности, в нем используется функция `isalpha()`, проверяющая буквенные символы, `isdigits()`, проверяющая символы десятичных цифр, таких как 3, `isspace()`, проверяющая пробельные символы, такие как пробелы, символы новой строки и табуляции, и `ispunct()`, проверяющая знаки препинания.

Программа также предлагает повторный обзор структуры `if else if` и цикла `while` с `cin.get(char)`.

Листинг 6.8. cctypes.cpp

```
// cctypes.cpp -- использование библиотеки ctype.h
#include <iostream>
#include <cctype>                    // прототипы символьных функций
int main()
{
   using namespace std;
   cout << "Enter text for analysis, and type @"
      " to terminate input.\n"; // запрос текста для анализа; завершающий символ - @
   char ch;
   int whitespace = 0;
   int digits = 0;
   int chars = 0;
   int punct = 0;
   int others = 0;
   cin.get(ch);                      // получение первого символа
   while (ch != '@')                 // проверка на признак окончания ввода
   {
            if(isalpha(ch))          // буквенный символ?
               chars++;
          else if(isspace(ch))       // пробельный символ?
             whitespace++;
      else if(isdigit(ch))           // десятичная цифра?
         digits++;
      else if(ispunct(ch))           // знак препинания?
         punct++;
      else
                        others++;
                     cin.get(ch);// получение следующего символа
   }
```

```
    cout << chars << " letters, "
         << whitespace << " whitespace, "
         << digits << " digits, "
         << punct << " punctuations, "
         << others << " others.\n";       // вывод количества букв, пробелов, цифр,
                                          // знаков препинания и прочих символов
    return 0;
}
```

Ниже показан пример выполнения программы из листинга 6.8 (обратите внимание, что символы новой строки относятся к пробельным):

```
Enter text for analysis, and type @ to terminate input.
AdrenalVision International producer Adrienne Vismonger
announced production of their new 3-D film, a remake of
"My Dinner with Andre," scheduled for 2013. "Wait until
you see the the new scene with an enraged Collossipede!"@
177 letters, 33 whitespace, 5 digits, 9 punctuations, 0 others.
```

В табл. 6.4 кратко описаны функции, доступные в пакете `cctype`. В некоторых системах присутствуют не все эти функции, а в некоторых могут существовать дополнительные функции.

Таблица 6.4. Символьные функции `cctype`

Имя функции	Возвращаемое значение
`isalnum()`	Возвращает `true`, если аргумент — буква или десятичная цифра
`isalpha()`	Возвращает `true`, если аргумент — буква
`isblank()`	Возвращает `true`, если аргумент — пробел или знак горизонтальной табуляции
`iscntrl()`	Возвращает `true`, если аргумент — управляющий символ
`isdigit()`	Возвращает `true`, если аргумент — десятичная цифра (0–9)
`isgraph()`	Возвращает `true`, если аргумент — любой печатаемый символ, отличный от пробела
`islower()`	Возвращает `true`, если аргумент — символ в нижнем регистре
`isprint()`	Возвращает `true`, если аргумент — любой печатаемый символ, включая пробел
`ispunct()`	Возвращает `true`, если аргумент — знак препинания
`isspace()`	Возвращает `true`, если аргумент — стандартный пробельный символ (т.е. пробел, прогон страницы, новая строка, возврат каретки, горизонтальная табуляция, вертикальная табуляция)
`isupper()`	Возвращает `true`, если аргумент — символ в верхнем регистре
`isxdigit()`	Возвращает `true`, если аргумент — шестнадцатеричная цифра (т.е. 0–9, a–f или A–F)
`tolower()`	Если аргумент — символ верхнего регистра, возвращает его вариант в нижнем регистре, иначе возвращает аргумент без изменений
`toupper()`	Если аргумент — символ нижнего регистра, возвращает его вариант в верхнем регистре, иначе возвращает аргумент без изменений

Операция ?:

Язык C++ включает операцию, которая часто может использоваться вместо оператора `if else`. Она называется *условной операцией*, записывается как `?:` и является единственной операцией C++, которая требует трех операндов. Ее общая форма выглядит следующим образом:

```
выражение1 ? выражение2 : выражение3
```

Если *выражение1* истинно, то значением всего условного выражения будет значение *выражение2*. В противном случае значением всего выражения будет *выражение3*. Ниже приведены два примера, демонстрирующие ее работу:

```
5 > 3 ? 10 : 12  // 5 > 3 истинно, поэтому значением всего выражения будет 10
3 == 9? 25 : 18  // 3 == 9 ложно, поэтому значением всего выражения будет 18
```

Первый пример можно перефразировать так: если 5 больше, чем 3, то выражение оценивается как 10; иначе оно оценивается как 12. В реальных ситуациях программирования выражения, конечно же, могут включать в себя переменные.

Код в листинге 6.9 использует условную операцию для нахождения большего из двух значений.

Листинг 6.9. `condit.cpp`

```
// condit.cpp -- использование условной операции
#include <iostream>
int main()
{
   using namespace std;
   int a, b;
   cout << "Enter two integers: ";      // запрос на ввод двух целых чисел
   cin >> a >> b;
   cout << "The larger of " << a << " and " << b;
   int c = a > b ? a : b;               // c = a, если a > b, иначе c = b
   cout << " is " << c << endl;         // вывод большего из указанных чисел
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 6.9:

```
Enter two integers: 25 28
The larger of 25 and 28 is 28
```

Ключевой частью программы является следующий оператор:

```
int c = a > b ? a : b;
```

Он выдает тот же результат, что и приведенные ниже операторы:

```
int c;
if (a > b)
   c = a;
else
   c = b;
```

По сравнению с последовательностью `if else` условная операция более короткая, но на первый взгляд не так очевидна. Одно отличие между этими двумя подходами заключается в том, что условная операция порождает выражение, а потому — единственное значение, которое может быть присвоено или встроено в более крупное

выражение, как это сделано в программе из листинга 6.9, где значение условного выражения присваивается переменной c. Краткая форма и необычный синтаксис условной операции высоко ценится некоторыми программистами. Их любимый трюк, достойный порицания, состоит в использовании вложенных друг в друга условных выражений, как показано в следующем относительно несложном примере:

```
const char x[2] [20] = {"Jason ","at your service\n"};
const char * y = "Quillstone ";
for (int i = 0; i < 3; i++)
   cout << ((i < 2) ? !i ? x[i] : y : x[1]);
```

Это лишь завуалированный (но отнюдь не максимально завуалированный) способ вывода трех строк в следующем порядке:

```
Jason Quillstone at your service
```

В терминах читабельности условная операция больше подходит для простых отношений и простых значений выражений вроде:

```
x = (x > y) ? x : y;
```

Если же код становится более сложным, то, возможно, более ясно его получится выразить с помощью оператора `if else`.

Оператор `switch`

Предположим, что вы создаете экранное меню, которое предлагает пользователю на выбор один из четырех возможных вариантов, например, "дешевый", "умеренный", "дорогой", "экстравагантный" и "непомерный". Вы можете расширить последовательность `if else if else` для обработки этих пяти альтернатив, но оператор C++ `switch` упрощает обработку выбора из большого списка. Ниже представлена общая форма оператора `switch`:

```
switch (целочисленное-выражение)
{
   case метка1 : оператор(ы)
   case метка2 : оператор(ы)
   ...
   default : оператор(ы)
}
```

Оператор `switch` действует подобно маршрутизатору, который сообщает компьютеру, какую строку кода выполнять следующей. По достижении оператора `switch` программа переходит к строке, которая помечена значением, соответствующим текущему значению *целочисленное-выражение*. Например, если *целочисленное-выражение* имеет значение `4`, то программа переходит к строке с меткой `case 4:`. Как следует из названия, выражение *целочисленное-выражение* должно быть целочисленным. Также каждая метка должна быть целым константным выражением. Чаще всего метки бывают константами типа `char` или `int`, такими как `1` или `'q'`, либо же перечислителями. Если *целочисленное-выражение* не соответствует ни одной метке, программа переходит к метке `default`. Метка `default` не обязательна. Если она опущена, а соответствия не найдено, программа переходит к оператору, следующему за `switch` (рис. 6.3).

Оператор `switch` в C++ отличается от аналогичных операторов в других языках, например, Pascal, в одном очень важном отношении. Каждая метка `case` в C++ работает только как метка строки, а не граница между выборами.

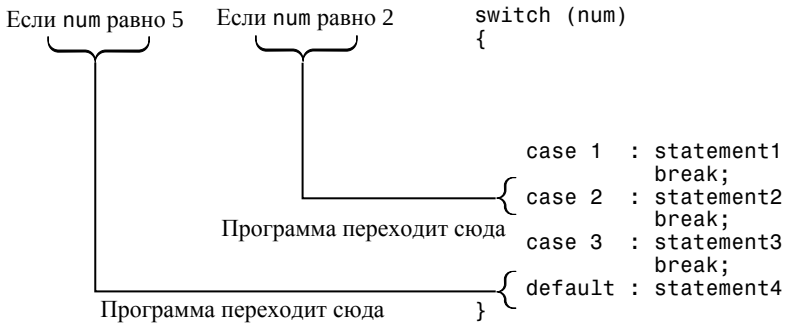


***Рис. 6.3.** Структура оператора* `switch`

То есть после того, как программа перейдет на определенную строку в `switch`, она последовательно выполнит все операторы, следующие за этой строкой внутри `switch`, если только вы явно не направите ее в другое место. Выполнение *не* останавливается автоматически на следующем `case`. Чтобы прекратить выполнение в конце определенной группы операторов, вы должны использовать оператор `break`. Это передаст управление за пределы блока `switch`.

В листинге 6.10 показано, как с помощью `switch` и `break` реализовать простое меню. Для отображения возможных вариантов выбора в программе применяется функция `showmenu()`. Затем оператор `switch` выбирает действие на основе выбора пользователя.

На заметку!

В некоторых комбинациях оборудования и операционной системы управляющая последовательность `\a` (см. `case 1` в листинге 6.10) не генерирует звуковой сигнал.

Листинг 6.10. `switch.cpp`

```
// switch.cpp -- использование оператора switch
#include <iostream>
using namespace std;
void showmenu();  // прототипы функций
void report();
void comfort();
int main()
{
   showmenu();
   int choice;
   cin >> choice;
   while (choice != 5)
   {
      switch(choice)
      {
         case 1 : cout << "\a\n";
                  break;
         case 2 : report();
                  break;
         case 3 : cout << "The boss was in all day.\n";
                  break;
         case 4 : comfort();
                  break;
         default : cout << "That's not a choice.\n";
      }
```

```
        showmenu();
        cin >> choice;
    }
    cout << "Bye!\n";
    return 0;
}

void showmenu()
{
    cout << "Please enter 1, 2, 3, 4, or 5:\n"
            "1) alarm 2)       report\n"
            "3) alibi 4)       comfort\n"
            "5) quit\n";
}

void report()
{
    cout << "It's been an excellent week for business.\n"
            "Sales are up 120%. Expenses are down 35%.\n";
}

void comfort()
{
    cout << "Your employees think you are the finest CEO\n"
            "in the industry. The board of directors think\n"
            "you are the finest CEO in the industry.\n";
}
```

Ниже показан пример выполнения программы из листинга 6.10:

```
Please enter 1, 2, 3, 4, or 5:
1) alarm         2) report
3) alibi         4) comfort
5) quit
4
Your employees think you are the finest CEO
in the industry. The board of directors think
you are the finest CEO in the industry.
Please enter 1, 2, 3, 4, or 5:
1) alarm         2) report
3) alibi         4) comfort
5) quit
2
It's been an excellent week for business.
Sales are up 120%. Expenses are down 35%.
Please enter 1, 2, 3, 4, or 5:
1) alarm         2) report
3) alibi         4) comfort
5) quit
6
That's not a choice.
Please enter 1, 2, 3, 4, or 5:
1) alarm         2) report
3) alibi         4) comfort
5) quit
5
Bye!
```

Цикл while завершается, когда пользователь вводит **5**. Ввод от **1** до **4** активизирует соответствующий выбор из списка switch, а ввод значения **6** вызывает действие по умолчанию.

Обратите внимание, что для корректной работы программы вводиться должно целочисленное значение. Если, например, ввести букву, оператор ввода даст сбой, а цикл будет выполняться бесконечно вплоть до уничтожения программы. В таком случае лучше использовать символьный ввод.

Как отмечалось ранее, этой программе необходимы операторы break, чтобы ограничить выполнение определенной частью оператора switch. Вы можете удостовериться, что это именно так, удалив операторы break из листинга 6.10 и посмотрев, как программа будет работать без них. Например, вы обнаружите, что ввод **2** заставит программу выполнить *все* операторы, ассоциированные с метками 2, 3, 4 и default. C++ ведет себя подобным образом, потому что такое поведение иногда может быть полезным. Так, за счет этого можно легко использовать множественные метки. Например, предположим, что вы перепишете листинг 6.10, используя в качестве выборов меню и меток case символы вместо целых значений. В таком случае можно использовать символы как верхнего, так и нижнего регистра:

```
char choice;
cin >> choice;
while (choice != 'Q' && choice != 'q')
{
   switch(choice)
   {
      case 'a':
      case 'A': cout << "\a\n";
              break;
      case 'r':
      case 'R': report();
              break;
      case 'l':
      case 'L': cout << "The boss was in all day.\n";
              break;
      case 'c':
      case 'C': comfort();
              break;
      default : cout << "That's not a choice.\n";
   }
   showmenu();
   cin >> choice;
}
```

Поскольку сразу за case 'a' не следует break, управление программой передается следующей строке, которая является оператором, следующим за case 'A'.

Использование перечислителей в качестве меток

В листинге 6.11 иллюстрируется применение enum для определения набора взаимосвязанных констант в операторе switch. В общем случае входной поток cin не распознает перечислимые типы (он не может знать, как вы определите их), поэтому программа читает выбор как int. Когда оператор switch сравнивает значение int с перечислимой меткой case, он приводит перечисление к типу int. Точно также перечисления приводятся к int в проверочном условии цикла while.

Листинг 6.11. enum.cpp

```
// enum.cpp -- использование enum
#include <iostream>
// создание именованный констант для значений 0 - 6
enum {red, orange, yellow, green, blue, violet, indigo};
int main()
{
   using namespace std;
   cout << "Enter color code (0-6): "; // ввод кода цвета
   int code;
   cin >> code;
   while (code >= red && code <= indigo)
   {
      switch (code)
      {
         case red    : cout << "Her lips were red.\n"; break;
         case orange : cout << "Her hair was orange.\n"; break;
         case yellow : cout << "Her shoes were yellow.\n"; break;
         case green  : cout << "Her nails were green.\n"; break;
         case blue   : cout << "Her sweatsuit was blue.\n"; break;
         case violet : cout << "Her eyes were violet.\n"; break;
         case indigo : cout << "Her mood was indigo.\n"; break;
      }
      cout << "Enter color code (0-6): ";
      cin >> code;
   }
   cout << "Bye\n";
   return 0;
}
```

Ниже показан пример вывода программы из листинга 6.11:

```
Enter color code (0-6): 3
Her nails were green.
Enter color code (0-6): 5
Her eyes were violet.
Enter color code (0-6): 2
Her shoes were yellow.
Enter color code (0-6): 8
Bye
```

Операторы switch и if else

Оба оператора — `switch` и `if else` — позволяют выбирать из списка альтернатив. Однако `if else` из них является более гибким оператором. Например, он позволяет обрабатывать диапазоны, как показано в следующем примере:

```
if (age > 17 && age < 35)
   index = 0;
else if (age >= 35 && age < 50)
   index = 1;
else if (age >= 50 && age < 65)
   index = 2;
else
   index = 3;
```

В отличие от этого, оператор switch не позволяет обрабатывать диапазоны. Каждая метка case оператора switch должна быть представлена одиночным значением. К тому же значение должно быть целым (что включает char), поэтому оператор switch не может проверять значения с плавающей точкой. К тому же значение метки case должно быть константой. Если вам необходимо проверять диапазоны, выполнять проверку значений с плавающей точкой или сравнивать две переменные, то вам следует использовать if else.

Если же, однако, все альтернативы могут быть идентифицированы целочисленными константами, то вы можете применять как switch, так и if else. А поскольку это та ситуация, для обработки которой специально был спроектирован оператор switch, его применение в этом случае более эффективно в смысле размера кода и скорости выполнения, если только речь не идет всего о паре возможных альтернатив выбора.

Совет

Если в конкретном случае можно использовать либо оператор switch, либо последовательность if else if, то обычная практика состоит в применении switch, когда имеется три или более альтернатив.

Операторы break и continue

Операторы break и continue позволяют программе пропускать часть кода. Оператор break можно использовать в операторе switch и в любых циклах. Он вызывает немедленную передачу управления за пределы текущего оператора switch или цикла. Оператор continue применяется только в циклах и вынуждает программу пропустить остаток тела цикла и сразу начать следующую итерацию (рис. 6.4).

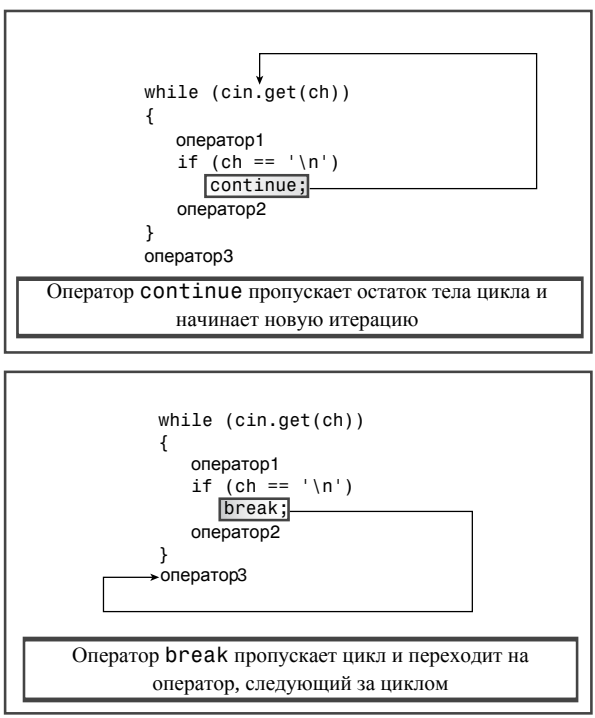


***Рис. 6.4.** Структура операторов* `break` *и* `continue`

В листинге 6.12 демонстрируется работа этих двух операторов. Программа позволяет ввести строку текста. Цикл отображает каждый ее символ и использует `break`, чтобы завершить цикл, если очередной символ строки окажется точкой. Это показывает, как с помощью `break` прервать цикл изнутри, если некоторое условие окажется истинным. Далее программа подсчитывает пробелы, пропуская остальные символы. Здесь в цикле используется `continue`, чтобы пропустить оставшуюся часть цикла, если окажется, что символ не является пробелом.

Листинг 6.12. `jump.cpp`

```
// jump.cpp -- использование операторов continue и break
#include <iostream>
const int ArSize = 80;
int main()
{
   using namespace std;
   char line[ArSize];
   int spaces = 0;
   cout << "Enter a line of text:\n";              // запрос на ввод строки текста
   cin.get(line, ArSize);
   cout << "Complete line:\n" << line << endl;     // вывод полной строки
   cout << "Line through first period:\n";         // вывод строки до первой точки
   for (int i = 0; line[i] != '\0'; i++)
   {
      cout << line[i];                            // отображение символа
      if (line[i] == '.')                         // завершение, если это точка
         break;
      if (line[i] != ' ')                         // пропуск оставшейся части цикла
         continue;
      spaces++;
   }
   cout << "\n" << spaces << " spaces\n";
   cout << "Done.\n";
   return 0;
}
```

Ниже приведен пример выполнения программы из листинга 6.12:

```
Enter a line of text:
Let's do lunch today. You can pay!
Complete line:
Let's do lunch today. You can pay!
Line through first period:
Let's do lunch today.
3 spaces
Done.
```

Замечания по программе

Обратите внимание, что в то время как оператор `continue` вынуждает программу из листинга 6.12 пропустить оставшуюся часть тела цикла, он не пропускает выражение обновления цикла. В цикле `for` оператор `continue` заставляет программу перейти непосредственно к выражению обновления, а затем — к проверочному выражению. В цикле `while`, однако, `continue` заставляет программу сразу выполнить проверочное условие. Поэтому любое обновляющее выражение в теле цикла `while`, которое следует за `continue`, будет пропущено. В некоторых случаях это может приводить к проблемам.

В этой программе можно было бы обойтись без continue. Вместо этого можно было бы использовать следующий код:

```
if (line[i] == ' ')
   spaces++;
```

Однако оператор continue может сделать программу более читабельной, когда за continue следует несколько операторов. В таком случае нет необходимости делать эти операторы частью if.

В C++, как и в C, присутствует оператор goto. Следующий оператор означает, что нужно перейти в место, помеченное меткой paris::

```
goto paris;
```

То есть в программе может присутствовать следующий код:

```
char ch;
cin >> ch;
if (ch == 'P')
   goto paris;
cout << ...
...
paris: cout << "You've just arrived at Paris.\n";
```

В большинстве случаев (некоторые скажут — во всех случаях) применение goto — неудачное решение, и для управления потоком выполнения программы вы должны стараться применять структурные управляющие конструкции вроде if else, switch, continue и т.п.

Циклы для чтения чисел

Предположим, что вы разрабатываете программу, которая должна читать последовательность чисел в массив. Вы хотите предоставить пользователю возможность прекращения ввода до заполнения массива. Один из способов сделать это — воспользоваться поведением cin. Рассмотрим следующий код:

```
int n;
cin >> n;
```

Что произойдет, если пользователь ответит на запрос, введя слово вместо числа? При таком несоответствии произойдут четыре события.

- Значение n не изменится.
- Некорректный ввод останется во входной очереди.
- Будет выставлен флаг ошибки в объекте cin.
- Результат вызова метода cin, будучи преобразованным к типу bool, даст значение false.

Тот факт, что метод вернет false, означает возможность использования нечислового ввода для прекращения цикла чтения чисел. Выставление флага ошибки cin из-за нечислового ввода означает, что вы должны сбросить этот флаг перед тем, как программа снова сможет читать ввод. Метод clear(), который также сбрасывает условие конца файла (end-of-file — EOF) (см. главу 5), позволяет сбросить флажок некорректного ввода. (Как некорректный ввод, так и EOF могут привести к тому, что cin вернет false. В главе 17 показано, как различать эти ситуации.) Рассмотрим несколько примеров, иллюстрирующих эти приемы.

Предположим, что требуется написать программу, которая вычисляет средний вес ежедневного улова рыбы. Допустим, что в день ловится максимум пять рыб, поэтому массива из пяти элементов достаточно для помещения всех данных, но бывает, что ловится меньше пяти рыб. В листинге 6.13 используется цикл, который прекращается, когда массив полон, или когда вы вводите что-то, отличное от числа.

Листинг 6.13. cinfish.cpp

```
// cinfish.cpp -- нечисловой ввод прекращает выполнение цикла
#include <iostream>
const int Max = 5;
int main()
{
   using namespace std;
   // получение данных
   double fish[Max];
   cout << "Please enter the weights of your fish.\n";
   cout << "You may enter up to " << Max
       << " fish <q to terminate>.\n";   // ввод веса пойманных рыб
   cout << "fish #1: ";
   int i = 0;
   while (i < Max && cin >> fish[i]) {
      if (++i < Max)
         cout << "fish #" << i+1 << ": ";
   }
   // вычисление среднего значения
   double total = 0.0;
   for (int j = 0; j < i; j++)
   total += fish[j];
   // вывод результатов
   if (i == 0)
      cout << "No fish\n";              // рыбы нет
   else
      cout << total / i << " = average weight of "
          << i << " fish\n";            // средний вес рыбы
   cout << "Done.\n";
   return 0;
}
```

На заметку!

Как упоминалось ранее, некоторые исполняющие среды требуют дополнительного кода для сохранения окна в открытом состоянии, чтобы можно было просмотреть вывод. Поскольку ввод `'q'` в этом примере отключает дальнейший ввод, обработка будет более сложной:

```
if (!cin)          // ввод прекращается с помощью нечислового значения
{
   cin.clear();    // сбор ввода
   cin.get();      // чтение q
}
cin.get();         // чтение конца строки после последнего ввода
cin.get();         // ожидание нажатия пользователем клавиши <Enter>
```

Если требуется, чтобы программа принимала ввод после завершения цикла, можно было бы также воспользоваться кодом, подобным приведенному в листинге 6.13.

В листинге 6.14 представлена дальнейшая иллюстрация применения возвращаемого значения `cin` и сброса `cin`.

Выражение `cin >> fish[i]` в листинге 6.13 — это на самом деле вызов функции-метода `cin`, которая возвращает сам объект `cin`. Если `cin` используется как часть проверочного условия, он преобразуется в тип `bool`. Преобразованное значение равно `true`, если ввод прошел успешно, и `false` — в противном случае. Значение `false` прекращает цикл. Ниже показан пример запуска этой программы:

```
Please enter the weights of your fish.
You may enter up to 5 fish <q to terminate>.
fish #1: 30
fish #2: 35
fish #3: 25
fish #4: 40
fish #5: q
32.5 = average weight of 4 fish
Done.
```

Обратите внимание на следующую строку кода:

```
while (i < Max && cin >> fish[i]) {
```

Вспомните, что в C++ правая часть логического выражения "И" не вычисляется, если левая часть дает `false`. В данном случае вычисление правой части означает использование `cin` для помещения ввода в массив. Если `i` равно `Max`, цикл прекращается без попыток чтения значений в позицию за пределами массива.

В предыдущем примере не предпринимается попытка читать информацию после получения нечислового ввода. Рассмотрим случай, когда это все-таки нужно делать. Предположим, что требуется ввести ровно пять результатов игры в гольф в программу, вычисляющую средний результат. Если пользователь вводит нечисловое значение, программа должна напомнить ему, что требуется число. Предположим, что программа обнаружила неправильный ввод пользователя. Она должна выполнить три действия.

1. Сбросить состояние `cin` для принятия нового ввода.
2. Освободиться от некорректного ввода.
3. Предложить пользователю повторить ввод.

Обратите внимание, что программа должна сбросить `cin` перед тем, как отклонять неверный ввод. В листинге 6.14 показано, как все это сделать.

Листинг 6.14. `cingolf.cpp`

```
// cingolf.cpp -- нечисловой ввод пропускается
#include <iostream>
const int Max = 5;
int main()
{
    using namespace std;
    // Получение данных
    int golf[Max];
    cout << "Please enter your golf scores.\n";
    cout << "You must enter " << Max << " rounds.\n"; // ввод результатов в гольфе
    int i;
    for (i = 0; i < Max; i++)
    {
        cout << "round #" << i+1 << ": ";
        while (!(cin >> golf[i])) {
            cin.clear();     // сброс ввода
```

```
            while (cin.get() != '\n')
                continue;                    // отбрасывание некорректного ввода
            cout << "Please enter a number: ";
        }
    }

    // Вычисление среднего
    double total = 0.0;
    for (i = 0; i < Max; i++)
        total += golf[i];

    // Вывод результатов
    cout << total / Max << " = average score "
         << Max << " rounds\n";
    return 0;
}
```

Ниже показан пример запуска программы из листинга 6.14:

```
Please enter your golf scores.
You must enter 5 rounds.
round #1: 88
round #2: 87
round #3: must i?
Please enter a number: 103
round #4: 94
round #5: 86
91.6 = average score 5 rounds
```

Замечания по программе

Центральная часть кода обработки ошибок в листинге 6.14 выглядит следующим образом:

```
while (!(cin >> golf[i])) {
    cin.clear();                         // сброс ввода
    while (cin.get() != '\n')
        continue;                        // отбрасывание некорректного ввода
    cout << "Please enter a number: ";
}
```

Если пользователь введет **88**, то выражение `cin` равно `true` и значение помещается в массив. Более того, поскольку `cin` равно `true`, выражение `!(cin >> golf[i])` принимает значение `false`, и внутренний цикл прекращается. Но если пользователь вводит **must i?**, то выражение `cin` принимает значение `false`, в массив ничего не помещается, выражение `!(cin >> golf[i])` принимает значение `true`, и программа входит во вложенный цикл `while`.

Первый оператор в цикле использует метод `clear()` для очистки ввода. Если этот оператор опустить, программа не сможет прочитать никакого нового ввода. Далее программа использует `cin.get()` в цикле `while`, чтобы прочитать остаток ввода до конца строки. Это позволяет удалить некорректный ввод вместе со всем, что может еще содержаться в строке.

Другой подход заключается в чтении до следующего пробела, что позволит удалять по одному слову за раз, вместо того, чтобы удалять всю некорректную строку. После этого программа напоминает пользователю, что нужно вводить число.

Простой файловый ввод-вывод

Иногда ввод с клавиатуры — не самый лучший выбор. Например, предположим, что вы пишете программу для анализа биржевой активности и загрузили файл, содержащий 1000 цен на акции. Было бы намного удобнее иметь программу, которая прочитает этот файл напрямую, нежели вводить все значения вручную. Точно так же было бы удобно, чтобы программа записывала свой вывод в файл, сохраняя результаты для последующего использования.

К счастью, C++ дает возможность применить ваш опыт программирования клавиатурного ввода и экранного вывода (вместе это называется *консольным вводом-выводом*) к работе с файлами (*файловый ввод-вывод*). В главе 17 эта тема раскрывается более подробно, а пока что мы рассмотрим простейший ввод-вывод текстовых файлов.

Текстовый ввод-вывод и текстовые файлы

Давайте еще раз рассмотрим концепцию текстового ввода-вывода. Когда вы используете `cin` для ввода, программа рассматривает ввод в качестве последовательности байтов, интерпретируемых как код символа. Независимо от типа данных назначения, ввод начинается с символьных данных — т.е. текстовой информации. Объект `cin` затем берет на себя ответственность за преобразование текста в другие типы. Чтобы увидеть, как это работает, давайте рассмотрим, как разный код обрабатывает одну и ту же строку ввода.

Предположим, что имеется следующая простая входная строка:

```
38.5 19.2
```

Давайте посмотрим, как эта строка ввода обрабатывается `cin` при использовании с данными разных типов. Для начала попробуем тип `char`:

```
char ch;
cin >> ch;
```

Первый символ входной строки присваивается `ch`. В данном случае первый символ — десятичная цифра `3`, и двоичный код этой цифры помещается в `ch`. Введенное значение и целевая переменная имеют тип `char`, поэтому никакого преобразования не требуется. (Обратите внимание, что сохраняется не числовое значение `3`, а код символа `'3'`.) После этого следующим символом во входной очереди является десятичная цифра `8`, и она же будет следующим символом, анализируемым следующей операцией ввода. Теперь попробуем тип `int` с теми же входными данными:

```
int n;
cin >> n;
```

В этом случае объект `cin` читает до первого нецифрового символа, т.е. он читает цифру `3` и цифру `8`, оставляя точку в качестве следующего символа во входной очереди. `cin` вычисляет, что эти два символа соответствуют числовому значению `38`, и двоичный код `38` копируется в `n`. Дальше попробуем тип `double`:

```
double x;
cin >> x;
```

В этом случае объект `cin` выполняет чтение до первого символа, который не может быть частью числа с плавающей точкой. То есть он читает цифру `3`, цифру `8`, символ точки и цифру `5`, оставляя пробел в качестве следующего символа во входной очереди. `cin` вычисляет, что эти четыре символа соответствует числовому значению `38.5`, и двоичный код (в формате плавающей точки) числа `38.5` копируется в `x`.

Теперь попробуем тип символьного массива:

```
char word[50];
cin >> word;
```

В этом случае cin читает до первого пробельного символа. То есть он читает цифру 3, цифру 8, символ точки, цифру 5, оставляя пробел во входной очереди. Объект cin помещает коды этих четырех символов в массив word и добавляет ограничивающий нулевой символ. Никаких преобразований не требуется.

И, наконец, попробуем другой вариант ввода для типа массива char:

```
char word[50];
cin.getline(word, 50);
```

Теперь cin читает вплоть до символа новой строки (строка входных данных содержит менее 50 символов). Все символы до заключительной цифры 2 помещаются в массив word, и к ним добавляется нулевой символ. Символ новой строки отбрасывается, и следующим символом во входной очереди будет первый символ после символа новой строки. Никакого преобразования не происходит.

При выводе происходит обратный процесс. То есть целые преобразуются в последовательности десятичных цифр, а числа с плавающей точкой — в последовательности десятичных цифр и ряда других символов (например, 284.53 или -1.587E+06). Символьные данные преобразования не требуют.

Основная идея состоит в том, что любой ввод начинается с текста. Поэтому файловый эквивалент консольного ввода — это текстовый файл, т.е. файл, в котором каждый байт хранит код некоторого символа. Не все файлы являются текстовыми. Например, базы данных и электронные таблицы хранят числовые данные в числовом же виде — в двоичной целочисленной форме или в двоичном представлении чисел с плавающей точкой. Также файлы, созданные текстовыми процессорами, могут хранить текстовую информацию, но они также содержат и нетекстовые данные, описывающие форматирование, шрифты, принтеры и т.п.

Файловый ввод-вывод в этой главе обсуждается параллельно с консольным вводом-выводом, а потому касается только текстовых файлов. Чтобы создать текстовый файл, вы используете текстовый редактор, такой как Notepad для Windows либо vi или emacs — для Unix/Linux. Можете также пользоваться текстовым процессором, но только сохранять файлы в текстовом формате. Редакторы кода, являющиеся частью интегрированных сред разработки (integrated development environment — IDE), также создают текстовые файлы; файлы с исходным кодом программ являются примерами текстовых файлов. Аналогичным образом можно применять текстовые редакторы для просмотра файлов, созданных текстовым выводом.

Запись в текстовый файл

Для файлового вывода в C++ используется аналог cout. Таким образом, чтобы подготовиться к файловому выводу, давайте рассмотрим ряд основных фактов относительно использования cout в консольном выводе.

- Вы должны включить заголовочный файл iostream.
- В заголовочном файле iostream определен класс ostream для обработки вывода.
- В заголовочном файле iostream объявлена переменная ostream, или объект по имени cout.

- Вы должны принимать в расчет пространство имен std; например, для таких элементов, как cout или endl, можно использовать директиву using или префикс std::.
- Можно использовать cout с операцией << для чтения данных различных типов.

Файловый вывод является очень похожей аналогией.

- Вы должны включить заголовочный файл fstream.
- В заголовочном файле fstream определен класс ofstream для обработки вывода.
- Вы должны объявить один или более переменных типа ofstream, или объектов, которые можете именовать по своему усмотрению, пока учитываете принятые для этого соглашения.
- Вы должны принимать в расчет пространство имен std; например, для таких элементов, как ofstream, можно использовать директиву using или префикс std::.
- Вы должны ассоциировать конкретный объект ofstream с определенным файлом; одним из способов сделать это является применение метода open().
- По окончании работы с файлом вы должны использовать метод close() для закрытия файла.
- Можно использовать ofstream с операцией << для чтения данных различных типов.

Следует отметить, что хотя заголовочный файл iostream представляет предопределенный объект ostream по имени cout, вы должны объявлять собственный объект ofstream, назначив ему имя и ассоциировав с файлом. Ниже показано, как объявляются такие объекты:

```
ofstream outFile;          // outFile – объект типа ofstream
ofstream fout;             // fout – объект типа ofstream
```

А вот как можно ассоциировать объекты с конкретными файлами:

```
outFile.open("fish.txt"); // outFile используется для записи в файл fish.txt
char filename[50];
cin >> filename;          // пользователь указывает имя файла
fout.open(filename);      // fout используется для чтения указанного файла
```

Обратите внимание, что метод open() требует в качестве аргумента строки в стиле C. Это может быть строковый литерал или строка, сохраненная в символьном массиве.

Ниже показано, как использовать эти объекты:

```
double wt = 125.8;
outFile << wt;             // записать число в fish.txt
char line[81] = "Objects are closer than they appear.";
fin << line << endl;       // записать строку текста
```

Важный момент заключается в том, что после объявления объекта ofstream и ассоциирования его с файлом он применяется точно так же, как cout. Все операции и методы, доступные cout, такие как <<, endl и setf(), также доступны для всех объектов ofstream, таких как outFile и fout в предыдущих примерах.

Короче говоря, ниже представлены основные шаги, связанные с использованием файлового вывода.

1. Включить заголовочный файл `fstream`.
2. Создать объект `ofstream`.
3. Ассоциировать объект `ofstream` с файлом.
4. Работать с объектом `ofstream` в той же манере, как с `cout`.

Этот подход демонстрируется в листинге 6.15. У пользователя запрашивается информация, вывод посылается сначала на экран, а затем в файл. Полученный файл можно просмотреть в текстовом редакторе.

Листинг 6.15. `outfile.cpp`

```
// outfile.cpp -- запись в файл
#include <iostream>
#include <fstream>  // для файлового ввода-вывода
int main()
{
   using namespace std;
   char automobile[50];
   int year;
   double a_price;
   double d_price;
   ofstream outFile;                                  // создание объекта для вывода
   outFile.open("carinfo.txt");                       // ассоциирование его с файлом
   cout << "Enter the make and model of automobile: "; // ввод производителя и модели
   cin.getline(automobile, 50);
   cout << "Enter the model year: ";                  // ввод года выпуска
   cin >> year;
   cout << "Enter the original asking price: ";       // ввод начальной цены
   cin >> a_price;
   d_price = 0.913 * a_price;

   // Отображение информации на экране с помощью cout
   cout << fixed;
   cout.precision(2);
   cout.setf(ios_base::showpoint);
   cout << "Make and model: " << automobile << endl;  // производитель и модель
   cout << "Year: " << year << endl;                  // год выпуска
   cout << "Was asking $" << a_price << endl;         // начальная цена
   cout << "Now asking $" << d_price << endl;         // конечная цена

   // Вывод той же информации с использованием outFile вместо cout
   outFile << fixed;
   outFile.precision(2);
   outFile.setf(ios_base::showpoint);
   outFile << "Make and model: " << automobile << endl;
   outFile << "Year: " << year << endl;
   outFile << "Was asking $" << a_price << endl;
   outFile << "Now asking $" << d_price << endl;
   outFile.close();                                   // завершить работу с файлом
   return 0;
}
```

Обратите внимание, что заключительный раздел программы в листинге 6.15 дублирует раздел `cout`, но вместо `cout` применяется `outFile`. Ниже приведен пример запуска этой программы:

```
Enter the make and model of automobile: Flitz Perky
Enter the model year: 2009
Enter the original asking price: 13500
Make and model: Flitz Perky
Year: 2009
Was asking $13500.00
Now asking $12325.50
```

Экранный вывод обеспечивается cout. Если вы проверите каталог или папку, которая содержит исполняемую программу, то найдете там новый файл carinfo.txt. Он содержит вывод, сгенерированный с помощью outFile. Если открыть его в текстовом редакторе, то в нем обнаружится следующее содержимое:

```
Make and model: Flitz Perky
Year: 2009
Was asking $13500.00
Now asking $12325.50
```

Как видите, outFile отправляет точно ту же последовательность символов в файл carinfo.txt, что cout посылает на экран.

Замечания по программе

После объявления объекта ofstream в программе из листинга 6.15 с помощью метода open() можно ассоциировать объект с определенным файлом:

```
ofstream outFile;                    // создание объекта для вывода
outFile.open("carinfo.txt");         // ассоциирование его с файлом
```

Когда работа с файлом завершена, соединение должно быть закрыто:

```
outFile.close();
```

Обратите внимание, что метод close() не требует передачи имени файла. Дело в том, что outFile уже был ассоциирован с конкретным файлом. Если вы забудете закрыть файл, программа закроет его автоматически при нормальном завершении.

Следует отметить, что outFile может использовать те же методы, что и cout. Он может применять не только операцию <<, но также разнообразные методы форматирования, такие как setf() и precision(). Эти методы влияют только на объект, который их вызывает. Например, можно указывать разные значения точности для разных объектов:

```
cout.precision(2);        // использовать точность 2 для вывода на экран
outFile.precision(4);     // использовать точность 4 для файлового вывода
```

Главное, что вы должны помнить — после установки такого объекта ofstream, как outFile, его можно использовать точно так же, как и стандартный cout.

Вернемся к методу open():

```
outFile.open("carinfo.txt");
```

В этом случае файл carinfo.txt перед запуском программы не существует. То есть здесь метод open() создает новый файл с таким именем.

Но если файл carinfo.txt уже существует, что случится, если вы запустите программу вновь? По умолчанию open() первым делом усечет его до нулевой длины, уничтожив старое содержимое. Затем содержимое будет заменено новым выводом.

В главе 17 описано, как можно переопределить это поведение по умолчанию.

Внимание!

Когда вы открываете существующий файл для вывода, по умолчанию он усекается до нулевой длины и его старое содержимое теряется.

Бывает, что попытка открыть файл для вывода не удается. Например, файл с указанным именем может уже существовать и иметь ограничения доступа. Поэтому внимательный программист должен проверить, удалась ли попытка открытия. Мы продемонстрируем необходимые для этого приемы в следующем примере.

Чтение текстового файла

Теперь рассмотрим файловый ввод. Он основан на консольном вводе, с которым связано множество аспектов. Начнем с их перечисления.

- Вы должны включить заголовочный файл `iostream`.
- В заголовочном файле `iostream` определее класс `istream` для обработки ввода.
- В заголовочном файле `iostream` объявлена переменная типа `istream`, или объект по имени `cin`.
- Вы должны принимать в расчет пространство имен `std`; например, для таких элементов, как `cin`, можно использовать директиву `using` или префикс `std::`.
- Вы можете использовать `cin` с операцией >> для чтения данных разнообразных типов.
- Вы можете использовать `cin` с методом `get()` для чтения индивидуальных символов и с методом `getline()` для чтения целых строк символов за раз.
- Вы можете использовать `cin` с такими методами, как `eof()` и `fail()`, чтобы отслеживать успешность попыток ввода.
- Сам объект `cin`, когда присутствует в проверочных условиях, преобразуется в булевское значение `true`, если последняя попытка чтения была успешной, и `false` — в противном случае.

Файловый ввод является очень похожей аналогией.

- Вы должны включить заголовочный файл `fstream`.
- В заголовочном файле `fstream` определен класс `ifstream` для обработки ввода.
- Вы должны объявить одну или более переменных `ifstream`, или объектов, которые можно назвать по своему усмотрению, учитывая принятые для этого соглашения.
- Вы должны принимать в расчет пространство имен `std`; например, для таких элементов, как `ifstream`, можно использовать директиву `using` или префикс `std::`.
- Вы должны ассоциировать конкретный объект `ifstream` с конкретным файлом; один из способов сделать это — воспользоваться методом `open()`.
- По завершении работы с файлом должен быть вызван метод `close()` для его закрытия.
- Вы можете использовать объект `ifstream` с операцией >> для чтения данных различных типов.
- Вы можете использовать объект `ifstream` с методом `get()` для чтения отдельных символов и с методом `getline()` — для чтения целых строк.

- Вы можете использовать объект ifstream с такими методами, как eof() и fail(), чтобы отслеживать успешность попыток ввода.
- Сам объект ifstream, когда присутствует в проверочных условиях, преобразуется в булевское значение true, если последняя попытка чтения была успешной, и false — в противном случае.

Следует отметить, что хотя заголовочный файл iostream представляет предопределенный объект istream по имени cin, вы должны объявлять собственный объект ifstream, выбирая для него имя и ассоциируя с файлом. Вот как объявляются такие объекты:

```
ifstream inFile;          // inFile - объект типа ifstream
ifstream fin;             // fin - объект типа ifstream
```

Ниже показано, как с ними можно ассоциировать конкретные файлы:

```
inFile.open("bowling.txt"); // inFile используется для чтения файла bowling.txt
char filename[50];
cin >> filename;          // имя файла указывает пользователь
fin.open(filename);       // fin используется для чтения указанного файла
```

Метод open() требует в качестве аргумента строки в стиле C. Это может быть литеральная строка или же строка, сохраненная в символьном массиве. Примеры использования этих объектов приведены ниже:

```
double wt;
inFile >> wt;             // чтение числа из bowling.txt
char line[81];
fin.getline(line, 81);    // чтение строки текста
```

Важный момент, который следует отметить: после объявления объекта ifstream и ассоциирования его с определенным файлом можно использовать его точно так же, как cin. Все операции и методы, доступные cin, также доступны объектам ifstream, как это демонстрировалось с применением inFile и fin в предыдущих примерах.

Что случится, если вы попытаетесь открыть для ввода несуществующий файл? Эта ошибка приведет к тому, что все последующие попытки использования объекта ifstream для ввода будут обречены на провал. Предпочтительный способ проверки того, удалось ли открыть файл, заключается в применении метода is_open(). Для этого можно применить следующий код:

```
inFile.open("bowling.txt");
if (!inFile.is_open())
{
   exit(EXIT_FAILURE);
}
```

Метод is_open() возвращает true, если файл открыт успешно, поэтому выражение !inFile.is_open() дает в результате true, если попытка оказывается неудачной. Прототип функции exit() находится в заголовочном файле cstdlib, где также определена константа EXIT_FAILURE как значение аргумента, используемого для взаимодействия программы с операционной системой. Функция exit() завершает программу.

Метод is_open() является относительно новым в C++. Если ваш компилятор не поддерживает его, можете воспользоваться вместо него методом good(). Как будет показано в главе 17, метод good() не проверяет возможные проблемы настолько тщательно, как это делает is_open().

Программа в листинге 6.16 открывает файл, указанный пользователем, читает из файла числа, после чего сообщает количество прочитанных значений, их сумму и среднюю величину. Здесь важно правильно спроектировать входной цикл, что обсуждается подробно в разделе "Замечания по программе". Обратите внимание на интенсивное использование в программе операторов `if`.

Листинг 6.16. `sumafile.cpp`

```
// sumfile.cpp -- чтение файла
#include <iostream>
#include <fstream>  // поддержка файлового ввода-вывода
#include <cstdlib>  // поддержка exit()
const int SIZE = 60;
int main()
{
   using namespace std;
   char filename[SIZE];
   ifstream inFile;                    // объект для обработки файлового ввода
   cout << "Enter name of data file: "; // запрос имени файла данных
   cin.getline(filename, SIZE);
   inFile.open(filename);              // ассоциирование inFile с файлом
   if (!inFile.is_open())              // не удалось открыть файл
   {
      cout << "Could not open the file " << filename << endl;
      cout << "Program terminating.\n";
      exit(EXIT_FAILURE);
   }
   double value;
   double sum = 0.0;
   int count = 0;                      // количество прочитанных элементов
   inFile >> value;                    // ввод первого значения
   while (inFile.good())               // пока ввод успешен и не достигнут EOF
   {
      ++count;                         // еще один элемент прочитан
      sum += value;                    // вычисление текущей суммы
      inFile >> value;                 // ввод следующего значения
   }
   if (inFile.eof())
      // достигнут конец файла
      cout << "End of file reached.\n";
   else if (inFile.fail())
      // ввод прекращен из-за несоответствия типа данных
      cout << "Input terminated by data mismatch.\n";
   else
      // ввод прекращен по неизвестной причине
      cout << "Input terminated for unknown reason.\n";
   if (count == 0)
      // данные для обработки отсутствуют
      cout << "No data processed.\n";
   else
   {
      cout << "Items read: " << count << endl;      // прочитано элементов
      cout << "Sum: " << sum << endl;               // сумма
      cout << "Average: " << sum / count << endl;   // среднее значение
   }
   inFile.close();                                  // завершение работы с файлом
   return 0;
}
```

Чтобы использовать программу из листинга 6.16, сначала понадобится создать текстовый файл, содержащий числа. Для этого можно воспользоваться текстовым редактором, который обычно применяется для подготовки исходных текстов программ. Предположим, что файл называется `scores.txt` и содержит следующие данные:

```
18 19 18.5 13.5 14
16 19.5 20 18 12 18.5
17.5
```

Программа должна иметь возможность найти этот файл. Обычно, если только при вводе имени файла не указывается также и путь, программа будет искать файл в том же каталоге, где хранится исполняемый файл.

Внимание!

В текстовом файле Windows для завершения строки текста используется символ возврата каретки, за которым следует символ перевода строки. (При чтении из файла в стандартном текстовом режиме C++ эта комбинация транслируется в символ новой строки, а при записи в файл выполняется обратное преобразование.) Некоторые текстовые редакторы, такие как встроенный редактор IDE-среды Metrowerks CodeWarrior, не добавляют автоматически эту комбинацию к последней строке файла. Поэтому, если вы пользуетесь таким редактором, то должны нажимать клавишу <Enter> после ввода последней строки текста и перед завершением файла.

Ниже показан пример запуска программы из листинга 6.16:

```
Enter name of data file: scores.txt
End of file reached.
Items read: 12
Sum: 204.5
Average: 17.0417
```

Замечания по программе

Вместо жесткого кодирования имени файла программа из листинга 6.16 сохраняет введенное пользователем имя в символьном массиве `filename`. Затем этот массив передается в качестве аргумента `open()`:

```
inFile.open(filename);
```

Как говорилось ранее в настоящей главе, весьма желательно проверять, удалась ли попытка открытия файла. Вот несколько причин возможных неудач: файл может не существовать, он может находиться в другом каталоге, к нему может быть запрещен доступ, либо пользователь может допустить опечатку при вводе его имени или не указать расширение. Многие начинающие программисты тратят массу времени, пытаясь разобраться, почему неправильно работает цикл чтения из файла, тогда как реальная проблема заключается в том, что программе просто не удалось его открыть. Проверка успешности открытия файла может сэкономить немало времени и усилий.

Правильному проектированию цикла чтения файла должно уделяться особое внимание. Есть несколько вещей, которые нужно проверять при чтении файла. Во-первых, программа не должна пытаться читать после достижения EOF. Метод `eof()` возвращает `true`, когда последняя попытка чтения данных столкнулась с EOF. Во-вторых, программа может столкнуться с несоответствием типа.

Например, программа из листинга 6.16 ожидает файла, содержащего только числа. Метод `fail()` возвращает `true`, когда последняя попытка чтения сталкивается с несоответствием типа. (Этот метод также возвращает `true` при достижении EOF.)

И, наконец, что-то другое может пойти не так — например, файл окажется поврежденным или произойдет сбой оборудования. Метод `bad()` вернет `true`, если самая последняя попытка чтения столкнется с такой проблемой. Вместо того чтобы проверять все эти условия индивидуально, проще применить метод `good()`, который возвращает `true`, если все идет хорошо:

```
while (inFile.good())     // пока ввод успешен и не достигнут EOF
{
   ...
}
```

Затем при желании можно воспользоваться другими методами для определения точной причины прекращения цикла:

```
if (inFile.eof())
   // Достигнут конец файла
   cout << "End of file reached.\n";
else if (inFile.fail())
   // Ввод прекращен из-за несоответствия типа данных
   cout << "Input terminated by data mismatch.\n";
else
   // Ввод прекращен по неизвестной причине
   cout << "Input terminated for unknown reason.\n";
```

Этот код находится сразу после цикла, поэтому он исследует, почему цикл был прерван. Поскольку `eof()` проверяет только EOF, а `fail()` проверяет как EOF, так и несоответствие типа, здесь вначале проверятся именно EOF. Таким образом, если выполнение дойдет до `else if`, то достижение конца файла (EOF) как причина выхода из цикла будет исключена, и значение `true`, полученное от `fail()`, недвусмысленно укажет на несоответствие типа.

Важно также понимать, что `good()` сообщает лишь о самой последней попытке чтения ввода. Это значит, что попытка чтения должна *непосредственно* предшествовать вызову `good()`. Стандартный способ обеспечения этого — иметь один оператор ввода непосредственно перед началом цикла, перед первой проверкой условия цикла, а второй — в конце цикла, непосредственно перед следующей проверкой условия цикла:

```
// Стандартная структура цикла чтения
inFile >> value;          // ввод первого значения
while (inFile.good())     // пока ввод успешен и не достигнут EOF
{
   // Тело цикла
   inFile >> value;       // ввод следующего значения
}
```

Код можно несколько сократить, используя тот факт, что показанное ниже выражение возвращает собственно `inFile`, а этот `inFile`, помещенный в контекст, в котором ожидается значение `bool`, вычисляется как `inFile.good()` — т.е. как `true` или `false`:

```
inFile >> value
```

Поэтому два оператора ввода можно заменить одним, используя его в качестве условия цикла. Это значит, что предыдущую структуру цикла можно заменить следующей:

```
// Сокращенная структура цикла чтения
// Ввод перед циклом опущен
while (inFile >> value)  // чтение и проверка успешности
{
   // Тело цикла
   // Ввод в конце цикла опущен
}
```

Эта структура по-прежнему следует принципу попытки чтения перед проверкой, поскольку для оценки выражения `inFile >> value` программа сначала пытается выполнить его, читая число в переменную `value`.

Теперь вы знакомы с основами файлового ввода-вывода.

Резюме

Программы и сам процесс программирования становятся более интересными, когда вводятся операторы, которые позволяют программе выбирать альтернативные действия. В C++ имеются операторы `if`, `if else` и `switch`, представляющие собой средства управления выбором пути выполнения.

Оператор `if` позволяет программе выполнить оператор или блок операторов в случае удовлетворения некоторого условия. То есть программа выполняет этот оператор или блок, только если конкретное условие истинно.

Оператор `if else` позволяет программе выбрать для выполнения один из двух операторов или блоков. Для представления последовательности вариантов выбора можно добавлять дополнительные операторы `if else`.

Оператор `switch` направляет поток управления программы в определенное место из списка возможных.

В C++ также доступны операции, помогающие принимать решения. В главе 5 рассматривались выражения отношений, которые сравнивают два значения.

В операторах `if` и `if else` в качестве проверочных условий обычно используются выражения сравнения. С помощью логических операций C++ (`&&`, `||` и `!`) можно комбинировать или модифицировать выражения сравнения для организации более сложных проверок. Условная операция (`?:`) предлагает компактный способ выбора одного из двух значений по условию.

Библиотека символьных функций `cctype` предоставляет удобный и мощный набор инструментов для анализа символьного ввода.

Циклы и операторы выбора являются полезными инструментами для организации файлового ввода-вывода, который во многом повторяет консольный. После объявления объектов `ifstream` и `ofstream` и ассоциирования их с файлами эти объекты можно использовать в той же манере, что и стандартные `cin` и `cout`.

Благодаря циклам и операторам для принятия решений C++, появляется возможность писать интересные, интеллектуальные и высокопроизводительные программы. Но мы только начали исследовать реальную мощь языка C++. Далее мы обратимся к функциям.

Вопросы для самоконтроля

1. Посмотрите на следующие два фрагмента кода для подсчета пробелов и переводов строк:

```
// Версия 1
while (cin.get(ch)) // завершение по eof
{
if (ch == ' ')
   spaces++;
if (ch == '\n')
   newlines++;
}

// Версия 2
while (cin.get(ch)) // завершение по eof
{
if (ch == ' ')
   spaces++;
else if (ch == '\n')
   newlines++;
}
```

   Какие преимущества (если они есть) у второй формы перед первой?

2. Какой эффект даст замена в листинге 6.2 выражения `++ch` на `ch+1`?

3. Внимательно изучите следующую программу:

```
#include <iostream>
using namespace std;
int main()
{
   char ch;
   int ct1, ct2;
   ct1 = ct2 = 0;
   while ((ch = cin.get()) != '$')
   {
      cout << ch;
      ct1++;
      if (ch = '$')
         ct2++;
      cout << ch;
   }
   cout <<"ct1 =" << ct1 << ", ct2 =" << ct2 << "\n";
   return 0;
}
```

   Предположим, что вы вводите следующие строки, нажимая клавишу <Enter> в конце каждой строки:

```
Hi!
Send $10 or $20 now!
```

   Каким будет вывод? (Вспомните, что ввод буферизуется.)

4. Постройте логические выражения для представления перечисленных ниже условий.

   а. `weight` больше или равно `115`, но меньше `125`.

   б. `ch` равно `q` или `Q`.

   в. `x` — четное, но не равно `26`.

   г. `x` — четное, но не кратно `26`.

   д. `donation` находится в диапазоне 1000–2000 или `guest` равно `1`.

   е. `ch` — буква в нижнем или верхнем регистре. (Предполагается, что буквы нижнего регистра кодируются последовательно и буквы верхнего регистра также кодируются последовательно, но между буквами нижнего и верхнего регистров имеется промежуток.)

5. В английском языке предложение "I will not not speak" означает то же, что и "I will speak". Является ли выражение `!!x` в языке C++ тем же самым, что и `x`?

6. Постройте условное выражение, которое эквивалентно абсолютному значению переменной. То есть если значение `x` положительное, то значением выражения будет просто `x`, но если значение `x` отрицательное, то значением выражения должно быть `-x`, которое является положительным.

7. Перепишите следующий фрагмент с применением `switch`:

```
if (ch == 'A')
   a_grade++;
else if (ch == 'B')
   b_grade++;
else if (ch == 'C')
   c_grade++;
else if (ch == 'D')
   d_grade++;
else
   f_grade++;
```

8. Каково преимущество в листинге 6.10 использования символьных меток, таких как `a` и `c`, вместо цифр для выбора в меню и в операторе `switch`? (Подсказка: подумайте о том, что произойдет, если пользователь введет `q` в любом регистре, и что случится, когда он введет в любом регистре `5`.)

9. Посмотрите на следующий фрагмент кода:

```
int line = 0;
char ch;
while (cin.get(ch))
{
   if (ch == 'Q')
      break;
   if (ch != '\n')
      continue;
   line++;
}
```

   Перепишите этот код так, чтобы в нем не использовались операторы `break` и `continue`.

Упражнения по программированию

1. Напишите программу, которая читает клавиатурный ввод до символа @ и повторяет его, за исключением десятичных цифр, преобразуя каждую букву верхнего регистра в букву нижнего регистра и наоборот. (Не забудьте о семействе функций cctype.)
2. Напишите программу, читающую в массив double до 10 значений пожертвований. (Или, если хотите, используйте шаблонный объект array.) Программа должна прекращать ввод при получении нечисловой величины. Она должна выдавать среднее значение полученных чисел, а также количество значений в массиве, превышающих среднее.
3. Напишите предшественник программы, управляемой меню. Она должна отображать меню из четырех пунктов, каждый из них помечен буквой. Если пользователь вводит букву, отличающуюся от четырех допустимых, программа должна повторно приглашать его ввести правильное значение до тех пор, пока он этого не сделает. Затем она должна выполнить некоторое простое действие на основе пользовательского выбора. Работа программы должна выглядеть примерно так:

```
Please enter one of the following choices:
c) carnivore   p) pianist
t) tree        g) game
f
Please enter a c, p, t, or g: q
Please enter a c, p, t, or g: t
A maple is a tree.
```

4. Когда вы вступите в Благотворительный Орден Программистов (БОП), к вам могут обращаться на заседаниях БОП по вашему реальному имени, по должности либо секретному имени БОП. Напишите программу, которая может выводить списки членов по реальным именам, должностям, секретным именам либо по предпочтению самого члена. В основу положите следующую структуру:

```
// Структура имен Благотворительного Ордена Программистов (БОП)
struct bop {
    char fullname[strsize];    // реальное имя
    char title[strsize];       // должность
    char bopname[strsize];     // секретное имя БОП
    int preference;            // 0 = полное имя, 1 = титул, 2 = имя БОП
};
```

В этой программе создайте небольшой массив таких структур и инициализируйте его соответствующими значениями. Пусть программа запустит цикл, который даст возможность пользователю выбирать разные альтернативы:

```
a. display by name      b. display by title
c. display by bopname   d. display by preference
q. quit
```

Обратите внимание, что "display by preference" (отображать по предпочтениям) не означает, что нужно отобразить член preference; это значит, что необходимо отобразить член структуры, который соответствует значению preference. Например, если preference равно 1, то выбор d должен вызвать отображение должности данного программиста. Пример запуска этой программы может выглядеть следующим образом:

```
Benevolent Order of Programmers Report
a. display by name      b. display by title
c. display by bopname   d. display by preference
q. quit
Enter your choice: a
Wimp Macho
Raki Rhodes
Celia Laiter
Hoppy Hipman
Pat Hand
Next choice: d
Wimp Macho
Junior Programmer
MIPS
Analyst Trainee
LOOPY
Next choice: q
Bye!
```

5. Королевство Нейтрония, где денежной единицей служит тварп, использует следующую шкалу налогообложения:

   первые 5 000 тварпов — налог 0%

   следующие 10 000 тварпов — налог 10%

   следующие 20 000 тварпов — налог 15%

   свыше 35 000 тварпов — налог 20%

   Например, если некто зарабатывает 38 000 тварпов, то он должен заплатить налогов 5000 × 0,00 + 10000 × 0,10 + 20000 × 0,15 + 3000 × 0,20, или 4 600 тварпов. Напишите программу, которая использует цикл для запроса доходов и выдачи подлежащего к выплате налога. Цикл должен завершаться, когда пользователь вводит отрицательное или нечисловое значение.

6. Постройте программу, которая отслеживает пожертвования в Общество Защиты Влиятельных Лиц. Она должна запрашивать у пользователя количество меценатов, а затем приглашать вводить их имена и суммы пожертвований от каждого. Информация должна сохраняться в динамически выделяемом массиве структур. Каждая структура должна иметь два члена: символьный массив (или объект `string`) для хранения имени и переменную-член типа `double` — для хранения суммы пожертвования. После чтения всех данных программа должна отображать имена и суммы пожертвований тех, кто не пожалел $10 000 и более. Этот список должен быть озаглавлен меткой "Grand Patrons". После этого программа должна выдать список остальных жертвователей. Он должен быть озаглавлен "Patrons". Если в одной из двух категорий не окажется никого, программа должна напечатать "none". Помимо отображения двух категорий, никакой другой сортировки делать не нужно.

7. Напишите программу, которая читает слова по одному за раз, пока не будет введена отдельная буква **q**. После этого программа должна сообщить количество слов, начинающихся с гласных, количество слов, начинающихся с согласных, а также количество слов, не попадающих ни в одну из этих категорий. Одним из возможных подходов может быть применение `isalpha()` для различения слов, начинающихся с букв, и остальных, с последующим применением `if` или

switch для идентификации тех слов, прошедших проверку isalpha(), которые начинаются с гласных. Пример запуска может выглядеть так:

```
Enter words (q to quit):
The 12 awesome oxen ambled
quietly across 15 meters of lawn. q
5 words beginning with vowels
4 words beginning with consonants
2 others
```

8. Напишите программу, которая открывает текстовый файл, читает его символ за символом до самого конца и сообщает количество символов в файле.

9. Выполните упражнение 6, но измените его так, чтобы данные можно было получать из файла. Первым элементом файла должно быть количество меценатов, а остальная часть состоять из пар строк, в которых первая строка содержит имя, а вторая — сумму пожертвования. То есть файл должен выглядеть примерно так:

```
4
Sam Stone
2000
Freida Flass
100500
Tammy Tubbs
5000
Rich Raptor
55000
```

7

Функции как программные модули C++

В ЭТОЙ ГЛАВЕ...

- Основы функций
- Прототипы функций
- Передача аргументов функциям по значению
- Проектирование функций для обработки массивов
- Использование параметров типа указателей `const`
- Проектирование функций для обработки текстовых строк
- Проектирование функций для обработки структур
- Проектирование функций для обработки объектов класса `string`
- Функции, вызывающие сами себя (рекурсия)
- Указатели на функции

Удовольствие можно найти в любой деятельности. Присмотритесь внимательнее и вы найдете его в функциях. Язык C++ сопровождается огромной библиотекой полезных функций (стандартная библиотека ANSI C плюс набор классов C++), но истинное удовольствие от программирования вам доставит написание собственных функций. (С другой стороны, реальное мастерство в программировании может быть обретено за счет более глубокого изучения того, что возможно делать с помощью библиотек STL и BOOST для C++.) В этой и следующей главах мы рассмотрим, как определять функции, как передавать им информацию и как ее из них получать. После небольшого обзора работы функций внимание в этой главе будет сосредоточено на том, как использовать функции с массивами, строками и структурами. В конце мы коснемся темы рекурсии и указателей на функции. Если вы имеете опыт программирования на C, то большая часть настоящей главы покажется знакомой. Но не поддавайтесь ложному ощущению, что здесь нет ничего нового. Язык C++ внес некоторые существенные дополнения к тому, что делали функции C, и в главе 8 мы рассмотрим их более подробно. А пока что обратимся к основам.

Обзор функций

Давайте посмотрим, что вы уже знаете о функциях. Для того чтобы использовать функцию в C++, вы должны выполнить следующие шаги:

- предоставить определение функции;
- представить прототип функции;
- вызвать функцию.

Если вы планируете пользоваться библиотечной функцией, то она уже определена и скомпилирована. К тому же вы можете, да и должны пользоваться стандартным библиотечным заголовочным файлом, чтобы предоставить своей программе доступ к прототипу. Все что вам остается — правильно вызвать эту функцию. В примерах, которые рассматривались до сих пор в настоящей книге, это делалось много раз. Например, перечень стандартных библиотечных функций C включает функцию `strlen()` для нахождения длины строки. Ассоциированный стандартный заголовочный файл `cstring` содержит прототип функции для `strlen()` и ряда других связанных со строками функций. Благодаря предварительной работе, выполненной создателями компилятора, вы используете `strlen()` без всяких забот.

Когда вы создаете собственные функции, то должны самостоятельно обработать все три аспекта — определение, прототипирование и вызов. В листинге 7.1 демонстрируются все три шага на небольшом примере.

Листинг 7.1. `calling.cpp`

```
// calling.cpp -- определение, прототипирование и вызов функции
#include <iostream>

void simple();           // прототип функции
int main()
{
   using namespace std;
   cout << "main() will call the simple() function:\n";
   simple();             // вызов функции
   cout << "main() is finished with the simple() function.\n";
   // cin.get();
   return 0;
}
```

```
// Определение функции
void simple()
{
   using namespace std;
   cout << "I'm but a simple function.\n";
}
```

Ниже показан вывод программы из листинга 7.1:

```
main() will call the simple() function:
I'm but a simple function.
main() is finished with the simple() function.
```

Выполнение программы в `main()` останавливается, как только управление передается функции `simple()`. По завершении `simple()` выполнение программы возобновляется в функции `main()`. В этом примере внутри каждого определения функции присутствует директива `using`, потому что каждая функция использует `cout`. В качестве альтернативы можно было бы поместить единственную директиву `using` над определением функции либо использовать `std::cout`.

Давайте рассмотрим перечисленные ниже шаги подробнее.

Определение функции

Все функции можно разбить на две категории: те, которые не возвращают значений, и те, которые их возвращают. Функции, не возвращающие значений, называются функциями типа `void` и имеют следующую общую форму:

```
void имяФункции(списокПараметров)
{
   оператор(ы)
   return;           // не обязательно
}
```

Здесь *списокПараметров* указывает типы и количество аргументов (параметров), передаваемых функции. Позднее в этой главе мы исследуем эту часть более подробно. Необязательный оператор `return` отмечает конец функции. При его отсутствии функция завершается на закрывающей фигурной скобке. Тип функции `void` соответствует процедуре в Pascal, подпрограмме FORTRAN, а также процедурам подпрограмм в современной версии BASIC. Обычно функция `void` используется для выполнения каких-то действий. Например, функция, которая должна напечатать слово "Cheers!" заданное число раз (`n`) может выглядеть следующим образом:

```
void cheers(int n)   // возвращаемое значение отсутствует
{
   for (int i = 0; i < n; i++)
      std::cout << "Cheers! ";
   std::cout << std::endl;
}
```

Параметр `int n` означает, что `cheers()` ожидает передачи значения типа `int` в качестве аргумента при вызове функции.

Функция с возвращаемым значением передает генерируемое ею значение функции, которая ее вызвала. Другими словами, если функция возвращает квадратный корень из `9.0` (`sqrt(9.0)`), то вызывающая ее функция получит значение `3.0`. Такая функция объявляется, как имеющая тот же тип, что и у возвращаемого ею значения.

Вот общая форма:

```
имяТипа имяФункции(списокПараметров)
{
    оператор(ы)
    return значение;     // значение приводится к типу имяТипа
}
```

Функции с возвращаемыми значениями требуют использования оператора `return` таким образом, чтобы вызывающей функции было возвращено значение. Само значение может быть константой, переменной либо общим выражением. Единственное требование — выражение должно сводиться по типу к *имяТипа* либо может быть преобразовано в *имяТипа*. (Если объявленным возвращаемым типом является, скажем, `double`, а функция возвращает выражение `int`, то `int` приводится к `double`.) Затем функция возвращает конечное значение в вызывавшую ее функцию. Язык C++ накладывает ограничения на типы возвращаемых значений: возвращаемое значение не может быть массивом. Все остальное допускается — целые числа, числа с плавающей точкой, указатели и даже структуры и объекты. (Интересно, что хотя функция C++ не может вернуть массив непосредственно, она все же может вернуть его в составе структуры или объекта.)

Как программист, вы не обязаны знать, каким образом функция возвращает значение, но это знание может существенно прояснить концепцию. Обычно функция возвращает значение, копируя его в определенный регистр центрального процессора либо в определенное место памяти. Затем вызывающая программа читает его оттуда. И возвращающая, и вызывающая функции должны подчиняться общему соглашению относительно типа данных, хранящихся в этом месте. Прототип функции сообщает вызывающей программе, что следует ожидать, а определение функции сообщает программе, что именно она возвращает (рис. 7.1). Предоставление одной и той же информации в прототипе и определении может показаться излишней работой, но это имеет глубокий смысл. Конечно, если вы хотите, чтобы курьер взял что-то с вашего рабочего стола в офисе, то вы увеличите шансы на правильное выполнение этой работы, если предоставите описание того, что требуется, как курьеру, так и кому-то, кто находится в офисе.

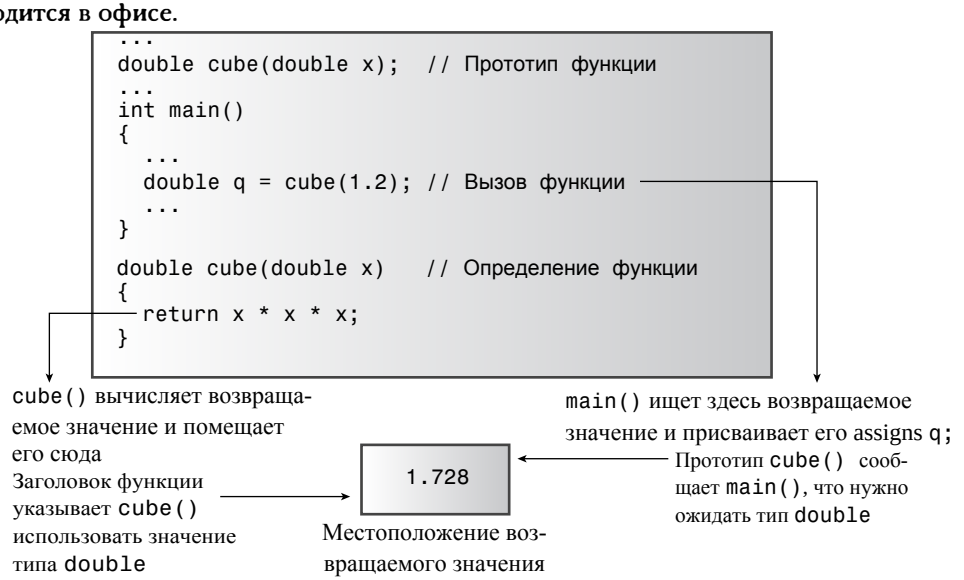


***Рис. 7.1.** Типичный механизм возврата значений*

Функция завершается после выполнения оператора `return`. Если функция содержит более одного оператора `return`, например, в виде альтернатив разных выборов `if else`, то в этом случае она прекращает свою работу по достижении первого оператора `return`. Например, в следующем коде конструкция `else` излишняя, однако она помогает понять намерение разработчика:

```
int bigger(int a, int b)
{
   if (a > b )
      return a;            // если a > b, функция завершается здесь
   else
      return b;            // в противном случае функция завершается здесь
}
```

(Обычно наличие в функции множества операторов `return` может сбивать с толку, и некоторые компиляторы предупреждают об этом. Тем не менее, приведенный выше код понять достаточно просто.)

Функции, возвращающие значения, очень похожи на функции в языках Pascal, FORTRAN и BASIC. Они возвращают значение вызывающей программе, которая затем может присвоить его переменной, отобразить на экране либо использовать каким-то другим способом. Ниже показан простой пример функции, которая возвращает куб значения типа `double`:

```
double cube(double x)       // x умножить на x и еще раз умножить на x
{
   return x * x * x;        // значение типа double
}
```

Например, вызов функции `cube(1.2)` вернет значение `1.728`. Обратите внимание, что здесь в операторе `return` находится выражение. Функция вычисляет значение выражения (в данном случае `1.728`) и возвращает его.

Прототипирование и вызов функции

Вы уже знакомы с тем, как вызываются функции, но, возможно, менее уверенно себя чувствуете в том, что касается их прототипирования, поскольку зачастую прототипы функций скрываются во включаемых (с помощью `#include`) файлах.

В листинге 7.2 демонстрируется использование функций `cheers()` и `cube()`; обратите внимание на их прототипы.

Листинг 7.2. `protos.cpp`

```
// protos.cpp -- использование прототипов и вызовы функций
#include <iostream>
void cheers(int);         // прототип: нет значения возврата
double cube(double x);    // прототип: возвращает double
int main()
{
   using namespace std;
   cheers(5);                       // вызов функции
   cout << "Give me a number: ";
   double side;
   cin >> side;
   double volume = cube(side);      // вызов функции
   cout << "A " << side <<"-foot cube has a volume of ";
   cout << volume << " cubic feet.\n";
   cheers(cube(2));                 // защита прототипа в действии
   return 0;
}
```

```
void cheers(int n)
{
    using namespace std;
    for (int i = 0; i < n; i++)
        cout << "Cheers! ";
    cout << endl;
}
double cube(double x)
{
    return x * x * x;
}
```

Программа из листинга 7.2 помещает директиву `using` только в те функции, которые используют члены пространства имен `std`. Вот пример запуска:

```
Cheers! Cheers! Cheers! Cheers! Cheers!
Give me a number: 5
A 5-foot cube has a volume of 125 cubic feet.
Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers! Cheers!
```

Обратите внимание, что `main()` вызывает функцию `cheers()` типа `void` с использованием имени функции и аргументов, за которыми следует точка с запятой: `cheers(5);`. Это пример оператора вызова функции. Но поскольку `cube()` возвращает значение, `main()` может применять его как часть оператора присваивания:

```
double volume = cube(side);
```

Но как говорилось ранее, необходимо сосредоточиться на прототипах. Что вы должны знать о прототипах? Для начала вы должны понять, почему C++ требует их. Затем, поскольку C++ требует прототипы, вам должен быть известен правильный синтаксис их написания. И, наконец, вы должны оценить, что они дают. Рассмотрим все эти вопросы по очереди, используя листинг 7.2 в качестве основы для обсуждения.

Зачем нужны прототипы?

Прототип описывает интерфейс функции для компилятора. Это значит, что он сообщает компилятору, каков тип возвращаемого значения, если оно есть у функции, а также количество и типы аргументов данной функции. Рассмотрим для примера, как влияет прототип на вызов функции в листинге 7.2:

```
double volume = cube(side);
```

Во-первых, прототип сообщает компилятору, что функция `cube()` должна принимать один аргумент типа `double`. Если программа не предоставит этот аргумент, то прототипирование позволит компилятору перехватить такую ошибку. Во-вторых, когда функция `cube()` завершает вычисление, она помещает возвращаемое значение в некоторое определенное место — возможно, в регистр центрального процессора, а может быть и в память. Затем вызывающая функция — `main()` в данном случае — извлекает значение из этого места. Поскольку прототип устанавливает, что `cube()` имеет тип `double`, компилятор знает, сколько байт следует извлечь и как их интерпретировать. Без этой информации он может только предполагать, а это то, чем заниматься он не должен.

Но вы все еще можете задаваться вопросом: зачем компилятору нужен прототип? Не может ли он просто заглянуть дальше в файл и увидеть, как определена функция? Одна из проблем такого подхода в том, что он не слишком эффективен. Компилятору пришлось бы приостановить компиляцию `main()` на то время, пока он прочитает остаток файла. Однако имеется еще более серьезная проблема: функция может и не

находиться в том же самом файле. Компилятор C++ позволяет разбивать программу на множество файлов, которые компилируются независимо друг от друга и позднее собираются вместе. В таком случае компилятор может вообще не иметь доступа к коду функции во время компиляции `main()`. То же самое справедливо и в ситуации, когда функция является частью библиотеки. Единственный способ избежать применения прототипа функции — поместить ее определение перед первым использованием. Это не всегда возможно. Кроме того, стиль программирования на C++ предусматривает размещение функции `main()` первой, поскольку это в общем случае предоставляет структуру программы в целом.

Синтаксис прототипа

Прототип функции является оператором, поэтому он должен завершаться точкой с запятой. Простейший способ получить прототип — скопировать заголовок функции из ее определения и добавить точку с запятой. Это, собственно, и делает программа из листинга 7.2 с функцией `cube()`:

```
double cube(double x);   // добавление ; к заголовку для получения прототипа
```

Однако прототип функции не требует предоставления имен переменных-параметров; достаточно списка типов. Программа из листинга 7.2 строит прототип `cheers()`, используя только тип аргумента:

```
void cheers(int);        // в прототипе можно опустить имена параметров
```

В общем случае в прототипе можно указывать или не указывать имена переменных в списке аргументов. Имена переменных в прототипе служат просто заполнителями, поэтому если даже они заданы, то не обязательно должны совпадать с именами в определении функции.

Сравнение прототипирования в C++ и ANSI C

Прототипирование ANSI C позаимствовано из C++, но его воплощение в этих двух языках отличается. Наиболее важно то, что в ANSI C из-за сохранения совместимости с классическим C прототипирование считается необязательным, в то время как в C++ прототипирование является обязательным. Например, рассмотрим следующее объявление функции:

```
void say_hi();
```

В C++ пустые скобки означают то же, что и указание ключевого слова `void` между ними. Это значит, что функция не имеет аргументов. В ANSI C пустые скобки означают просто, что список аргументов не указан. Другими словами, вы просто решили не прототипировать список аргументов. Эквивалентом отсутствия списка аргументов в C++ является многоточие:

```
void say_bye(...); // C++ отказывается от ответственности за список аргументов
```

Обычно такое применение многоточия необходимо только для взаимодействия с функциями C, такими как `printf()`, которые имеют переменное количество аргументов.

Что обеспечивают прототипы

Вы увидели, что прототипы помогают компиляторам. Но чем они полезны программистам? Прототипы значительно снижают вероятность допущения ошибок в программе. В частности, они обеспечивают следующие моменты.

- Компилятор корректно обрабатывает возвращаемое значение.
- Компилятор проверяет, указано ли правильное количество аргументов.
- Компилятор проверяет правильность типов аргументов. Если тип не подходит, компилятор преобразует его в правильный, когда это возможно.

Мы уже говорили о том, как корректно обработать возвращаемое значение. Теперь посмотрим, что случится, если вы зададите неверное количество аргументов.

Например, предположим, что в программе выполнен следующий вызов:

```
double z = cube();
```

Компилятор, в котором не используется прототипирование функций, пропустит это. Когда функция будет вызвана, он обнаружит, что `cube()` должна принимать число, и подставит любое подвернувшееся значение. Именно так работал С до того, как в ANSI С было позаимствовано прототипирование из С++. Поскольку прототипирование в ANSI С не обязательно, подобным образом некоторые программы С ведут себя и до сих пор. Но в С++ прототипирование обязательно, поэтому вы застрахованы от ошибок подобного рода.

Далее предположим, что вы предоставили аргумент, но неверного типа. В С это может приводить к возникновению странных ошибок. Например, если функция ожидает тип `int` (предположим, что он имеет размер 16 бит), а вы передали `double` (предположим, 64 бита), то функция видит только первые 16 бит из 64 и пытается интерпретировать их как значение типа `int`. Однако С++ автоматически преобразует переданное значение к типу, указанному в прототипе, предполагая, что оба типа арифметические. Например, в листинге 7.2 присутствуют два несоответствия типа в одном операторе:

```
cheers(cube(2));
```

Программа передает целое значение `2` функции `cube()`, которая ожидает тип `double`. Компилятор, замечая, что прототип `cube()` указывает тип аргумента `double`, преобразует `2` в `2.0`, т.е. в значение типа `double`. Затем `cube()` возвращает значение `8.0` типа `double`, которое должно быть использовано в качестве аргумента `cheers()`. Опять-таки, компилятор проверяет прототип и замечает, что `cheers()` требует аргумента `int`. Он преобразует возвращенное значение в целочисленное `8`. В общем случае прототипирование позволяет выполнять автоматическое приведение к ожидаемым типам. (Однако перегрузки функций, рассматриваемые в главе 8, могут породить неоднозначные ситуации, которые предотвращают выполнение определенных автоматических приведений типов.)

Автоматическое преобразование типов не позволяет исключить все возможные ошибки. Например, если вы передаете значение `8.33E27` в функцию, ожидающую аргумента `int`, то такое большое значение не может быть корректно преобразовано в обычный тип `int`. Некоторые компиляторы предупреждают о возможных потерях данных при автоматическом преобразовании больших типов в малые.

К тому же прототипирование позволяет выполнять преобразование типов только тогда, когда это имеет смысл. Например, невозможно преобразовать целое в структуру или указатель.

Прототипирование происходит во время компиляции и называется *статическим контролем типов*. Статический контроль типов, как вы уже видели, обнаруживает многие ошибки, которые было бы трудно перехватить во время выполнения.

Аргументы функций и передача по значению

Наступило время внимательнее взглянуть на аргументы функций. В С++ они обычно передаются *по значению*. Это означает, что числовое значение аргумента передается в функцию, где присваивается новой переменной.

Например, в листинге 7.2 присутствует следующий вызов функции:

```
double volume = cube(side);
```

Здесь `side` — переменная, которая в примере запуска получает значение `5`. Вспомним, что заголовок функции `cube()` был таким:

```
double cube(double x)
```

Когда эта функция вызывается, она создает новую переменную типа `double` по имени `x` и инициализирует ее значением `5`. Это позволяет изолировать данные в `main()` от того, что происходит в `cube()`, т.к. `cube()` работает с копией `side`, а не с исходными данными.

Вскоре вы увидите пример такой защиты. Переменная, которая используется для приема переданного значения, называется *формальным аргументом* или *формальным параметром*. Значение, переданное функции, называется *фактическим аргументом* или *фактическим параметром*. Чтобы немного упростить ситуацию, в стандарте C++ слово *аргумент* используется для обозначения фактического аргумента или параметра, а слово *параметр* — для обозначения формального аргумента или параметра. Применяя эту терминологию, можно сказать, что передача аргумента инициализирует параметр значением этого аргумента (рис. 7.2).

***Рис. 7.2.** Передача по значению*

Переменные, включая параметры, объявленные в функции, являются приватными по отношению к этой функции. Когда функция вызывается, компьютер выделяет память, необходимую для этих переменных. Когда функция завершается, компьютер освобождает память, которая была использована этими переменными. (В некоторых источниках по C++ это выделение и освобождение памяти для переменных называется *созданием и уничтожением переменных*. Возможно, это звучит более выразительно.) Такие переменные называются *локальными переменными*, потому что они локализованы в пределах функции.

Как уже упоминалось ранее, это помогает предохранить целостность данных. Это также означает, что если вы объявили переменную `x` в `main()` и другую переменную `x` в какой-то другой функции, то это будут две совершенно разных, никак не связанных друг с другом переменных (рис. 7.3). Такие переменные также называются *автоматическими переменными*, поскольку они размещаются и освобождаются автоматически во время выполнения программы.

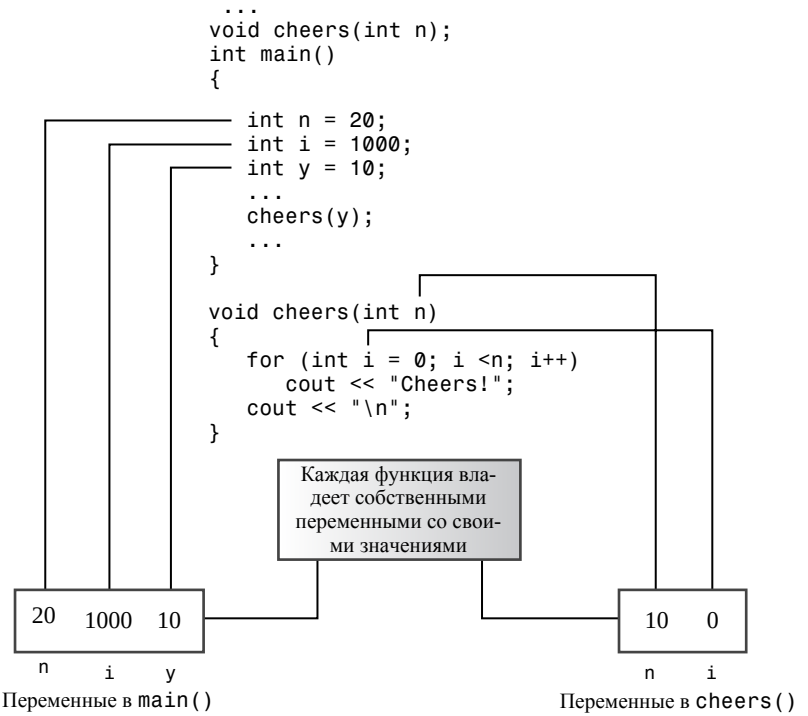


***Рис. 7.3.** Локальные переменные*

Множественные аргументы

Функция может принимать более одного аргумента. При вызове функции такие аргументы просто отделяются друг от друга запятыми:

```
n_chars('R', 25);
```

Это передает два аргумента функции `n_chars()`, определение которой будет приведено чуть позже.

Аналогично, при определении функции используется разделенный запятыми список параметров в ее заголовке:

```
void n_chars(char c, int n)    // два параметра
```

Этот заголовок устанавливает, что функция `n_chars()` принимает один параметр типа `char` и один типа `int`. Параметры `c` и `n` инициализируются значениями, переданными функции. Если функция имеет два параметра одного и того же типа, то типы каждого параметра должны указываться по отдельности. Комбинирование объявлений аргументов, как это делается с обычными переменными, не допускается:

```
void fifi(float a, float b)    // объявляет каждую переменную отдельно
void fufu(float a, b)          // не допускается
```

Как и в случае других функций, для получения прототипа нужно просто добавить точку с запятой:

```
void n_chars(char c, int n);  // прототип, стиль 1
```

Подобно ситуации с единственным аргументом, вы не обязаны использовать одинаковые имена переменных в прототипе и определении.

Также можно опускать имена переменных в прототипе:

```
void n_chars(char, int);       // прототип, стиль 2
```

Тем не менее, указание имен переменных часто помогает прояснить прототип, особенно если два параметра имеют один и тот же тип. Впоследствии имена параметров будут напоминать, какой аргумент для чего предназначен:

```
double melon_density(double weight, double volume);
```

В листинге 7.3 представлен пример функции с двумя аргументами. Он также иллюстрирует тот факт, что изменение значения формального параметра внутри функции никак не влияет на данные вызывающей программы.

Листинг 7.3. `twoarg.cpp`

```
// twoarg.cpp -- функция с двумя аргументами
#include <iostream>
using namespace std;
void n_chars(char, int);
int main()
{
   int times;
   char ch;
   cout << "Enter a character: ";        // ввод символа
   cin >> ch;
   while (ch != 'q')                     // q для завершения
   {
      cout << "Enter an integer: ";      // ввод целого числа
      cin >> times;
      n_chars(ch, times);                // функция с двумя аргументами
      cout << "\nEnter another character or press the"
           " q-key to quit: ";           // ввод другого символа или q для завершения
      cin >> ch;
   }
   cout << "The value of times is " << times << ".\n"; // вывод значения переменной times
   cout << "Bye\n";
   return 0;
}
void n_chars(char c, int n)              // вывод значения c n раз
{
   while (n-- > 0)                       // продолжение, пока n не достигнет 0
   cout << c;
}
```

Программа в листинге 7.3 иллюстрирует помещение директивы `using` над определением функции вместо ее использования внутри функции. Ниже показан пример выполнения:

```
Enter a character: W
Enter an integer: 50
WWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWWW
Enter another character or press the q-key to quit: a
Enter an integer: 20
aaaaaaaaaaaaaaaaaaaa
Enter another character or press the q-key to quit: q
The value of times is 20.
Bye
```

Замечания по программе

Функция `main()` в листинге 7.3 использует цикл `while` для организации повторяющегося ввода (и чтобы освежить ваши знания о циклах). Обратите внимание, что для чтения символа в ней применяется `cin >> ch`, а не `cin.get(ch)` или `ch = cin.get()`. На то имеется серьезная причина. Вспомните, что две функции `cin.get()` читают все входные символы, включая пробелы и символы новой строки, в то время как `cin >>` пропускает пробелы и символы новой строки. Когда вы отвечаете на приглашение к вводу в программе, то должны нажимать <Enter> в конце каждой строки, генерируя тем самым символ новой строки. Подход `cin >> ch` пропускает эти лишние символы, тогда как оба варианта `cin.get()` читают символ новой строки, следующий за каждым введенным числом, как очередной символ для отображения. Этот нюанс можно обойти программным путем, но проще использовать `cin`, как это делается в программе из листинга 7.3.

Функция `n_chars()` принимает два аргумента: символ `c` и целое число `n`. Затем в цикле она отображает символ столько раз, сколько указано в `n`:

```
while (n-- > 0)  // продолжение, пока n не достигнет 0
   cout << c;
```

Обратите внимание, что программа выполняет подсчет, уменьшая на каждом шаге значение переменной `n`, которая является формальным параметром из списка аргументов. Этой переменной присваивается значение переменной `times` в `main()`. Цикл `while` уменьшает `n` до `0`, но, как демонстрирует пример выполнения, изменение значения `n` никак не отражается на значении `times`. Даже если в `main()` вместо имени `times` использовать имя `n`, значение `n` в `main()` не затрагивается изменениями значения `n` в `n_chars()`.

Еще одна функция с двумя аргументами

Теперь давайте создадим более сложную функцию — такую, которая будет выполнять нетривиальные вычисления. К тому же помимо применения формальных параметров функция проиллюстрирует использование локальных переменных.

Сейчас многие штаты в США организуют различного рода лотереи. Эти лотереи предлагают выбрать определенные числа из многих, представленных на карточке. Например, вы можете выбрать 6 чисел в карточке, содержащей всего 51 число. Затем организаторы лотереи выбирают случайным образом 6 номеров. Если ваш вариант полностью совпал с тем, что выбрали организаторы, вы получаете несколько миллионов долларов или около того. Наша функция будет вычислять вероятность выигрыша. (Конечно, функция, которая могла бы успешно угадывать выигрышные номера, была бы более полезной, но язык C++, несмотря на всю его мощность, пока не может учитывать психологические факторы.)

Для начала нам понадобится формула. Если вы должны угадать 6 значений из 51, математики говорят, что у вас имеется один шанс выигрыша из R, где R вычисляется по следующей формуле:

$$R = \frac{51 \times 50 \times 49 \times 48 \times 47 \times 46}{6 \times 5 \times 4 \times 3 \times 2 \times 1}.$$

Для шести чисел в знаменателе будет произведение первых шести целых чисел, или 6!. Числитель же вычисляется как произведение шести последовательных чисел, на этот раз начинающихся с 51 и ниже. В общем, если нужно выбрать `picks` значений из `numbers` чисел, то числителем будет факториал для `picks`, а знаменателем — произведение `picks` целых чисел, начиная со значения `numbers` и ниже.

Для выполнения этого вычисления можно воспользоваться циклом for:

```
long double result = 1.0;
for (n = numbers, p = picks; p > 0; n--, p--)
   result = result * n / p ;
```

Вместо того чтобы сразу перемножить все составляющие числителя, цикл начинает с умножения 1.0 на первую составляющую числителя и делит его на первую составляющую знаменателя. Затем на следующем шаге цикл умножает и делит результат на следующие составляющие числителя и знаменателя. Это позволяет сохранять текущее произведение меньшим, чем если бы сначала выполнялось все умножение. Например, сравните

```
(10 * 9) / (2 * 1)
```

и

```
(10 / 2) * (9 / 1)
```

Первое выражение вычисляется как 90/2 и дает в результате 45, а второе вычисляется как 5 × 9 с получением того же результата 45. Результаты одинаковы, но в первом случае получается большее промежуточное значение (90), нежели во втором. Чем больше множителей, тем существеннее будет разница. Для больших чиссл эта стратегия замены умножения делением может предохранить процесс вычисления от переполнения максимально возможного значения с плавающей точкой.

В листинге 7.4 эта формула заключена в функцию probability(). Поскольку количество вариантов выбора и общее количество чисел должны быть положительными значениями, в программе для этих величин используется тип unsigned int (сокращенно — unsigned). Перемножение нескольких целых может породить достаточно большие результаты, поэтому в lotto.cpp для возвращаемого значения функции применяется тип long double. К тому же такие выражения, как 49/6, порождают ошибки округления при работе с целочисленными типами.

На заметку!

Некоторые реализации C++ не поддерживают тип long double. Если ваша реализация относится к ним, используйте просто double.

Листинг 7.4. lotto.cpp

```
// lotto.cpp -- вероятность выигрыша
#include <iostream>
// Примечание: некоторые реализации требуют применения double вместо long double
long double probability(unsigned numbers, unsigned picks);
int main()
{
   using namespace std;
   double total, choices;
   // Ввод общего количества номеров и количества номеров, которые нужно угадать
   cout << "Enter the total number of choices on the game card and\n"
           "the number of picks allowed:\n";
   while ((cin >> total >> choices) && choices <= total)
   {
      cout << "You have one chance in ";
      cout << probability(total, choices);     // вычисление и вывод шансов
      cout << " of winning.\n";
      cout << "Next two numbers (q to quit): ";
            // Ввод следующих двух чисел (q для завершения)
   }
```

```
    cout << "bye\n";
    return 0;
}

// Следующая функция вычисляет вероятность правильного
// угадывания picks чисел из numbers возможных
long double probability(unsigned numbers, unsigned picks)
{
    long double result = 1.0;  // несколько локальных переменных
    long double n;
    unsigned p;
    for (n = numbers, p = picks; p > 0; n--, p--)
        result = result * n / p ;
    return result;
}
```

Ниже показан пример выполнения программы из листинга 7.4:

```
Enter the total number of choices on the game card and
the number of picks allowed:
49 6
You have one chance in 1.39838e+007 of winning.
Next two numbers (q to quit): 51 6
You have one chance in 1.80095e+007 of winning.
Next two numbers (q to quit): 38 6
You have one chance in 2.76068e+006 of winning.
Next two numbers (q to quit): q
bye
```

Обратите внимание, что увеличение количества вариантов в игровой карточке существенно снижает шансы на выигрыш.

Замечания по программе

Функция `probability()` из листинга 7.4 иллюстрирует два вида локальных переменных, которые встречаются в функциях. Первый — это формальные параметры (`numbers` и `picks`), объявленные в заголовке функции внутри круглых скобок. Затем идут другие локальные переменные (`result`, `n` и `p`). Они объявлены между фигурными скобками, ограничивающими определение функции. Основная разница между формальными параметрами и другими локальными переменными состоит в том, что формальные параметры получают свои значения из функции, которая вызывает `probability()`, в то время как локальные переменные получают свои значения внутри функции.

Функции и массивы

До сих пор все примеры функций, приведенные в данной книге, были простыми и использовали для своих аргументов и возвращаемых значений только базовые типы. Однако функции могут служить инструментами и для обработки более сложных типов, таких как массивы и структуры. Давайте посмотрим, как соотносятся друг с другом массивы и функции.

Предположим, что вы используете массив, чтобы отследить, сколько печенья съел каждый участник семейного пикника. (Каждый индекс массива соответствует определенному лицу, а значение элемента — количеству съеденного печенья.) Необходим также общий итог. Его легко вычислить: нужно просто применить цикл для суммирования всех элементов массива. Но сложение элементов массива — настолько часто

встречающаяся операция, что имеет смысл спроектировать функцию для решения этой задачи. Тогда вам не придется писать новый цикл каждый раз, когда понадобится суммировать элементы массива.

Давайте посмотрим, как должен выглядеть интерфейс функции. Поскольку функция вычисляет сумму, она должна возвращать ответ. Если вы поглощаете печенье целиком, то можно использовать функцию с типом возврата `int`. Чтобы функция знала, какой массив суммировать, ей понадобится передавать в качестве аргумента имя массива. И чтобы не ограничивать ее массивами определенного размера, нужно будет также передавать ей размер массива. Единственный новый ингредиент здесь — это имя массива в качестве одного из формальных аргументов. Давайте посмотрим, что получилось:

```
int sum_arr(int arr[], int n)  // arr = имя массива, n = размер
```

Выглядит вполне правдоподобно. Квадратные скобки указывают на то, что `arr` — массив, а тот факт, что они пусты, говорит о том, что эту функцию можно применять с массивами любого размера. Но бывает, что некоторые вещи не являются тем, чем кажутся: `arr` — на самом деле не массив, а указатель! Однако хорошей новостью будет то, что остальную часть функции можно записать так, как если бы аргумент `arr` все-таки был массивом. Для начала убедимся на примере, что такой подход работает, а потом разберемся, почему.

В листинге 7.5 иллюстрируется применение указателя, как если бы он был именем массива. Программа инициализирует массив некоторыми значениями и затем использует функцию `sum_arr()` для вычисления суммы. Обратите внимание, что `sum_arr()` работает с `arr`, как если бы он был именем массива.

Листинг 7.5. `arrfun1.cpp`

```
// arrfun1.cpp -- функция с аргументом-массивом
#include <iostream>
const int ArSize = 8;
int sum_arr(int arr[], int n);  // прототип
int main()
{
   using namespace std;
   int cookies[ArSize] = {1,2,4,8,16,32,64,128};
   // Некоторые системы требуют предварить int словом static,
   // чтобы разрешить инициализацию массива
   int sum = sum_arr(cookies, ArSize);
   cout << "Total cookies eaten: " << sum << "\n"; // вывод количества съеденного печенья
   return 0;
}

// Возвращает сумму элементов массива целых чисел
int sum_arr(int arr[], int n)
{
    int total = 0;
    for (int i = 0; i < n; i++)
       total = total + arr[i];
    return total;
}
```

Вывод программы из листинга 7.5 выглядит следующим образом:

```
Total cookies eaten: 255
```

Как видите, программа работает. Теперь разберемся с тем, почему она работает.

Как указатели позволяют функциям обрабатывать массивы

Ключевым аспектом в программе 7.5 является то, что язык C++, подобно C, в большинстве контекстов трактует имя массива как указатель. Вспомните из главы 4, что имя массива интерпретируется как адрес его первого элемента:

```
cookies == &cookies[0]          // имя массива — это адрес его первого элемента
```

(Существует несколько исключений из этого правила. Во-первых, объявление массива использует имя массива в качестве метки хранилища. Во-вторых, применение операции `sizeof` к имени массива дает размер всего массива в байтах. В-третьих, как упоминалось в главе 4, применение операции взятия адреса `&` к имени массива позволяет получить адрес всего массива; например, `&cookies` будет адресом 32-байтного блока памяти при условии, что тип `int` занимает 4 байта.)

В листинге 7.5 присутствует следующий вызов функции:

```
int sum = sum_arr(cookies, ArSize);
```

Здесь `cookies` — имя массива, поэтому, согласно правилам C++, `cookies` представляет собой адрес первого элемента этого массива. То есть функции передается адрес. Поскольку массив имеет тип элементов `int`, аргумент `cookies` должен иметь тип указателя на `int`, или `int *`. Это предполагает, что правильный заголовок функции должен быть таким:

```
int sum_arr(int * arr, int n)  // arr = имя массива, n = размер
```

Здесь `int * arr` заменяет собой `int arr[]`. На самом деле оба варианта заголовка корректны, потому что в C++ нотации `int * arr` и `int arr[]` имеют идентичный смысл, когда (и *только* когда) применяются в заголовке или прототипе функции. Оба варианта означают, что `arr` — указатель на `int`. Но версия с нотацией массива (`int arr[]`) символически напоминает о том, что `arr` — не просто указатель на `int`, но указатель на первый `int` в массиве. В этой книге мы применяем нотацию массивов, когда указатель указывает на первый элемент в массиве, и нотацию указателей — когда имеется в виду указатель на отдельный элемент. Помните, однако, что в других контекстах нотации `int * arr` и `int arr[]` синонимами не являются. Например, вы не можете использовать нотацию `int tip[]` для объявления указателя в теле функции.

Принимая во внимание, что `arr` — на самом деле указатель, становится понятен смысл остальной части функции. Если вспомнить дискуссию о динамических массивах в главе 4, для обращения к индивидуальным элементам массива нотацию с квадратными скобками можно использовать одинаково хорошо как с именами массивов, так и с указателями. Будь `arr` указателем или именем массива, выражение `arr[3]` в любом случае означает четвертый элемент массива. Не помешает также напомнить о справедливости следующих утверждений:

```
arr[i] == *(ar + i)             // значения в двух нотациях
&arr[i] == ar + i               // адреса в двух нотациях
```

Вспомните, что добавление единицы к указателю, включая имя массива, на самом деле прибавляет к адресу значение размера в байтах того типа, на который указывает данный указатель. Увеличение указателя и применение индекса — это два эквивалентных способа подсчета элементов от начала массива.

Последствия использования массивов в качестве аргументов

Рассмотрим, что следует из листинга 7.5. Вызов функции `sum_arr(cookies, ArSize)` передает в эту функцию адрес первого элемента массива `cookies` и количе-

ство его элементов. Функция `sum_arr()` инициализирует адрес `cookies` указателем `arr`, а значение `ArSize` — переменной `n` типа `int`. Это значит, что в листинге 7.5 на самом деле в функцию не передается содержимое массива. Вместо этого программа сообщает функции, где находится массив (адрес), разновидность его элементов (тип) и сколько в нем содержится элементов (переменная `n`), как показано на рис. 7.4. Вооруженная этой информацией, функция затем использует исходный массив. Если вы передаете обычную переменную, то функция работает с ее копией. Но если вы передаете массив, то функция работает с оригиналом. В действительности эта разница не нарушает подхода передачи по значению, принятого в C++. Функция `sum_arr()` по-прежнему принимает значение, которое присваивается новой переменной. Но это значение является адресом, а не содержимым массива.

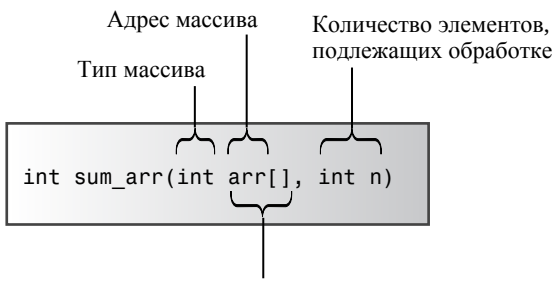


***Рис. 7.4.** Передача функции информации о массиве*

Хорошо ли то, что между именами массивов и указателями имеется соответствие? Безусловно. Проектное решение, связанное с использованием адресов массивов в качестве аргументов, позволит сэкономить время и память, необходимую для копирования всего массива. При работе с большими массивами накладные расходы, возникающие из-за использования таких копий, могли бы оказаться весьма ощутимыми. С копиями программам понадобилось бы не только больше компьютерной памяти, но и больше времени на копирование крупных блоков данных. С другой стороны, работа с исходными данными чревата возможностью непреднамеренного их повреждения. Это — реальная проблема в классическом C, но в ANSI C и C++ предусмотрен модификатор `const`, который обеспечивает необходимую защиту. Скоро вы увидите пример. Но сначала давайте изменим код в листинге 7.5, чтобы проиллюстрировать некоторые моменты, связанные с тем, как работают функции массивов. Программа в листинге 7.6 демонстрирует, что `cookie` и `arr` содержат одни и те же значения. Она также показывает, что концепция указателей делает функцию `sum_arr()` более изменчивой и гибкой, нежели могло показаться вначале. Чтобы внести немного разнообразия, для обеспечения доступа к `cout` и `endl` в программе применяется квалификатор `std::` вместо директивы `using`.

Листинг 7.6. `arrfun2.cpp`

```
// arrfun2.cpp -- функция с аргументом-массивом
#include <iostream>
const int ArSize = 8;
int sum_arr(int arr[], int n);
// использование std:: вместо директивы using
int main()
{
   int cookies[ArSize] = {1,2,4,8,16,32,64,128};
```

```
    // Некоторые системы требуют предварить int словом static,
    // чтобы разрешить инициализацию массива
    std::cout << cookies << " = array address, ";
    // Некоторые системы требуют приведения типа: unsigned (cookies)
    std::cout << sizeof cookies << " = sizeof cookies\n";
    int sum = sum_arr(cookies, ArSize);
    std::cout << "Total cookies eaten: " << sum << std::endl;
              // Общее количество съеденного печенья
    sum = sum_arr(cookies, 3);                          // первая хитрость
    std::cout << "First three eaters ate " << sum << " cookies.\n";
              // Съеденное первыми тремя
    sum = sum_arr(cookies + 4, 4);                      // вторая хитрость
    std::cout << "Last four eaters ate " << sum << " cookies.\n";
              // Съеденное последними четырьмя
    return 0;
}

// Возвращает сумму элементов целочисленного массива
int sum_arr(int arr[], int n)
{
    int total = 0;
    std::cout << arr << " = arr, ";
    // Некоторые системы требуют приведения типа: unsigned (arr)
    std::cout << sizeof arr << " = sizeof arr\n";
    for (int i = 0; i < n; i++)
        total = total + arr[i];
    return total;
}
```

Ниже показан вывод программы из листинга 7.6:

```
003EF9FC = array address, 32 = sizeof cookies
003EF9FC = arr, 4 = sizeof arr
Total cookies eaten: 255
003EF9FC = arr, 4 = sizeof arr
First three eaters ate 7 cookies.
003EFA0C = arr, 4 = sizeof arr
Last four eaters ate 240 cookies.
```

Обратите внимание, что значения адресов и размеров могут изменяться от системы к системе. К тому же, некоторые реализации отображают адреса в десятичной системе, а не в шестнадцатеричной. Другие реализации будут использовать шестнадцатеричные цифры и префикс `0x`.

Замечания по программе

Код в листинге 7.6 иллюстрирует некоторые очень интересные моменты, касающиеся функций, которые работают с массивами. Для начала обратите внимание, что `cookies` и `arr`, как и утверждалось, находятся по одному и тому же адресу в памяти. Но `sizeof cookies` равно 32, в то время как `sizeof arr` составляет 4. Причина в том, что `sizeof cookies` — размер всего массива, тогда как `sizeof arr` — размер переменной-указателя. (Программа выполнялась в системе с 4-байтными адресами.) Кстати, именно поэтому в `sum_arr()` нужно передавать размер массива вместо `sizeof arr`: указатель сам по себе не отражает размер массива.

Поскольку единственный способ для `sum_arr()` узнать количество элементов в массиве — через второй аргумент, вы можете схитрить.

Например, второй вызов функции выглядит следующим образом:

```
sum = sum_arr(cookies, 3);
```

Сообщая функции, что cookies имеет только три элемента, вы заставляете ее подсчитать сумму первых трех элементов.

Но зачем на этом останавливаться? Вы можете также схитрить относительно местоположения массива:

```
sum = sum_arr(cookies + 4, 4);
```

Поскольку cookies является адресом первого элемента, то cookies + 4 — адрес пятого элемента. Этот оператор суммирует пятый, шестой, седьмой и восьмой элементы массива. Следует отметить, что при третьем вызове функции передается другой адрес arr, отличный от того, что был передан в первых двух вызовах. Конечно же, в качестве аргумента можно было бы использовать &cookies[4] вместо cookies + 4 — оба варианта означают одно и то же.

На заметку!

Чтобы указать разновидность массива и количество элементов для функции, обрабатывающей массив, информация передается в двух отдельных аргументах:

```
void fillArray(int arr[], int size);    // прототип
```

Не пытайтесь передавать размер массива в нотации квадратных скобок:

```
void fillArray(int arr[size]);          // НЕТ — неудачный прототип
```

Дополнительные примеры функций для работы с массивами

Когда вы выбираете для представления данных массив, то тем самым принимаете проектное решение. Однако проектные решения не должны ограничиваться тем, как данные хранятся, они также должны учитывать, как они используются. Часто вы сочтете полезным написание специальных функций для выполнения специфических операций над данными. (Среди преимуществ такого подхода — повышение надежности программы, простота ее модификации и облегчение процесса отладки.) Также когда при обдумывании программы вы начинаете интеграцию средств хранения с операциями, то тем самым делаете важный шаг в сторону объектно-ориентированного образа мышления, что может принести существенные выгоды в будущем.

Рассмотрим простой случай. Предположим, что вы хотите использовать массив для отслеживания стоимости недвижимости. Вы должны решить, какой тип данных для этого использовать. Определенно double менее ограничен по диапазону допустимых значений, нежели int или long, и он предлагает достаточно значимых разрядов, чтобы точно представлять значения. Далее вы должны решить, сколько понадобится элементов. (В случае динамических массивов, созданных с помощью new, это решение можно отложить, но пока не будем усложнять.) Допустим, что будет не более пяти единиц недвижимости, поэтому можно использовать массив из пяти double.

Теперь посмотрим, какие операции может понадобиться выполнять над этим массивом. Двумя самыми главными являются чтение значений в массив и отображение его содержимого. Добавим в список еще одну операцию: переоценку стоимости объектов. Для простоты предположим, что объекты растут или падают в цене синхронно. (Помните, что это книга по C++, а не по операциям с недвижимостью.) Далее разработаем функцию для каждой операции и затем напишем соответствующий код. Итак, сначала выполним все шаги по разработке частей программы, а потом соберем их в единое целое.

Наполнение массива

Поскольку функция с аргументом — именем массива получает доступ к исходному массиву, а не к копии, вы можете использовать вызов функции для присваивания значений его элементам. Одним аргументом такой функции будет имя наполняемого массива. В общем случае программа может управлять инвестициями более чем одного лица, а потому в ней может быть более одного массива, следовательно, вы не захотите встраивать размер массива в саму функцию. Вместо этого вы передадите размер массива во втором аргументе, как это делалось в предыдущем примере. К тому же, возможно, понадобится прекратить чтение данных до того, как массив будет заполнен, поэтому такая возможность должна быть встроена в функцию. Так как допускается вводить меньше, чем максимальное количество элементов, имеет смысл обеспечить, чтобы функция возвращала действительное количество введенных значений. Все эти соглашения приводят к следующему прототипу:

```
int fill_array(double ar[], int limit);
```

Функция принимает аргумент с именем массива и аргумент, указывающий максимальное число элементов для чтения, а возвращает количество действительно введенных элементов. Например, если вы используете эту функцию с массивом из пяти элементов, то передаете во втором аргументе 5. Если затем вы вводите только три элемента, функция возвращает 3.

Для чтения в массив следующих друг за другом значений можно воспользоваться циклом, но как завершить этот цикл раньше? Одним из способов может быть применение специального значения для обозначения завершения ввода. Поскольку ни одно из вводимых значений не может быть отрицательным, то для указания конца ввода можно использовать отрицательное значение. К тому же функция должна както обрабатывать неправильный ввод, например, прекращая дальнейшие запросы значений. Учитывая все это, вы можете написать эту функцию следующим образом:

```
int fill_array(double ar[], int limit)
{
  using namespace std;
  double temp;
  int i;
  for (i = 0; i < limit; i++)
  {
    cout << "Enter value #" << (i + 1) << ":";    // ввод значения
    cin >> temp;
    if (!cin)                                     // неправильный ввод
    {
      cin.clear();
      while (cin.get() != '\n')
        continue;
      cout << "Bad input; input process terminated.\n"; // ввод прекращен
        break;
    }
    else if (temp < 0)                            // сигнал завершения
      break;
    ar[i] = temp;
  }
  return i;
}
```

Обратите внимание, что этот код включает выдачу приглашения пользователю на ввод. Если пользователь вводит неотрицательное значение, оно записывается в мас-

сив. В противном случае цикл прекращается. Если пользователь вводит только правильные значения, то цикл завершается после чтения limit значений. Последнее, что делает цикл – увеличивает i, поэтому после завершения цикла i равно величине, на единицу большей, чем последний индекс массива, т.е. i соответствует количеству введенных элементов. Затем функция возвращает это значение.

Отображение массива и защита его посредством `const`

Построить функцию для отображения содержимого массива очень просто. Ей передается имя массива и количество заполненных элементов, а она использует цикл отображения каждого из них. Но есть еще одно обстоятельство – нужно гарантировать, что функция отображения не внесет в исходный массив никаких изменений. Если только назначение функции не предусматривает внесения изменений в переданные ей данные, вы должны каким-то образом предохранить ее от этого. Такая защита обеспечивается автоматически для обычных аргументов, потому что C++ передает их по значению, и функция имеет дело с копиями. Но функция, работающая с массивом, обращается к оригиналу. В конце концов, именно поэтому предыдущая функция, fill_array(), в состоянии выполнять свою работу. Чтобы предотвратить случайное изменение содержимого массива-аргумента, при объявлении формального аргумента можно применить ключевое слово const (описанное в главе 3):

```
void show_array(const double ar[], int n);
```

Это объявление устанавливает, что указатель ar указывает на константные данные. Это значит, что использовать ar для изменения данных нельзя. То есть обратиться к такому значению, как к ar[0], можно, но изменить его не получится. Следует отметить, что это вовсе не означает, что исходный массив должен быть константным; это значит лишь, что вы не можете использовать ar внутри функции show_array() для изменения данных. Другими словами, show_array() трактует массив как данные, доступные только для чтения. Предположим, вы нечаянно нарушили это ограничение, попытавшись внутри функции show_array() сделать что-то вроде такого:

```
ar[0] += 10;
```

В этом случае компилятор пресечет ваши некорректные действия. Например, Borland C++ выдаст сообщение об ошибке такого вида:

```
Cannot modify a const object in function
   show_array(const double *,int)
Невозможно изменять константный объект в функции
   show_array(const double *,int)
```

Другие компиляторы могут выражать это иначе.

Сообщение подобного рода напоминает, что C++ интерпретирует объявление const double ar[] как const double *ar. Таким образом, это объявление действительно говорит о том, что ar указывает на константное значение. Мы поговорим об этом подробнее, когда завершим рассмотрение данного примера. А пока ниже приведен код функции show_array().

```
void show_array(const double ar[], int n)
{
   using namespace std;
   for (int i = 0; i < n; i++)
   {
      cout << "Property #" << (i + 1) << ": $";
      cout << ar[i] << endl;
   }
}
```

Изменение массива

Третья операция с массивом в этом примере — умножение каждого элемента на один и тот же коэффициент. Для ее выполнения функции придется передавать три аргумента: коэффициент, массив и количество элементов. Возвращать какое-либо значение не требуется, поэтому функция будет выглядеть следующим образом:

```
void revalue(double r, double ar[], int n)
{
    for (int i = 0; i < n; i++)
        ar[i] *= r;
}
```

Поскольку эта функция предназначена для изменения элементов массива, слово `const` в объявлении `ar` не указывается.

Собираем все вместе

Теперь, когда данные определены в терминах их хранения (массив), а также в терминах их использования (три функции), можно собрать вместе программу, использующую это проектное решение. Поскольку все операции управления массивом уже реализованы, это значительно упрощает программирование `main()`. Программа должна проверить, ввел ли пользователь число в ответ на запрос коэффициента переоценки. Вместо того чтобы останавливать выполнение в случае некорректного ввода, организуется цикл, запрашивающий у пользователя правильное значение коэффициента. Бо́льшая часть оставшейся работы по программированию сводится к вызовам разработанных функций в теле `main()`. В листинге 7.7 показан результат сборки всех частей. Директива `using` находится только в тех функциях, в которых применяются средства `iostream`.

Листинг 7.7. `arrfun3.cpp`

```
// arrfun3.cpp -- функция работы с массивами и применение const
#include <iostream>
const int Max = 5;
// Прототипы функций
int fill_array(double ar[], int limit);
void show_array(const double ar[], int n);     // не изменять данные
void revalue(double r, double ar[], int n);
int main()
{
    using namespace std;
    double properties[Max];
    int size = fill_array(properties, Max);
    show_array(properties, size);
    if (size > 0)
    {
        cout << "Enter revaluation factor: ";    // ввод коэффициента переоценки
        double factor;
        while (!(cin >> factor))                 // неправильный ввод
        {
          cin.clear();
          while (cin.get() != '\n')
              continue;
          cout << "Bad input; Please enter a number: "; // повторный запрос на ввод числа
        }
        revalue(factor, properties, size);
        show_array(properties, size);
    }
```

```
    cout << "Done.\n";
    cin.get();
    cin.get();
    return 0;
}
int fill_array(double ar[], int limit)
{
    using namespace std;
    double temp;
    int i;
    for (i = 0; i < limit; i++)
    {
      cout << "Enter value #" << (i + 1) << ": ";   // ввод значения
      cin >> temp;
      if (!cin)                                     // неправильный ввод
      {
        cin.clear();
        while (cin.get() != '\n')
           continue;
        cout << "Bad input; input process terminated.\n"; // процесс ввода прекращен
        break;
      }
      else if (temp < 0)                            // сигнал завершения
        break;
      ar[i] = temp;
    }
    return i;
}
// Следующая функция может использовать, но не изменять, массив по адресу ar
void show_array(const double ar[], int n)
{
    using namespace std;
    // Вывод содержимого массива ar
    for (int i = 0; i < n; i++)
    {
       cout << "Property #" << (i + 1) << ": $";
       cout << ar[i] << endl;
    }
}
// Умножает на r каждый элемент ar[]
void revalue(double r, double ar[], int n)
{
    for (int i = 0; i < n; i++)
       ar[i] *= r;
}
```

Ниже показаны два примера выполнения программы из листинга 7.7:

```
Enter value #1: 100000
Enter value #2: 80000
Enter value #3: 222000
Enter value #4: 240000
Enter value #5: 118000
Property #1: $100000
Property #2: $80000
Property #3: $222000
Property #4: $240000
Property #5: $118000
```

```
Enter revaluation factor: 0.8
Property #1: $80000
Property #2: $64000
Property #3: $177600
Property #4: $192000
Property #5: $94400
Done.

Enter value #1: 200000
Enter value #2: 84000
Enter value #3: 160000
Enter value #4: -2
Property #1: $200000
Property #2: $84000
Property #3: $160000
Enter reevaluation factor: 1.20
Property #1: $240000
Property #2: $100800
Property #3: $192000
Done.
```

Вспомните, что `fill_array()` предполагает прекращение ввода, когда пользователь введет пять элементов либо отрицательное значение — в зависимости от того, что произойдет раньше. Первый пример вывода иллюстрирует достижение предела по количеству элементов, а второй — прекращение приема значений по вводу отрицательной величины.

Замечания по программе

Мы уже обсудили важные детали программирования примера, поэтому обратимся к процессу в целом. Мы начали с обдумывания типа данных и разработали соответствующий набор функций для их обработки. Затем мы встроили эти функции в программу. Это то, что иногда называют *восходящим программированием*, поскольку процесс проектирования идет от частей-компонентов к целому. Этот подход хорошо стыкуется с объектно-ориентированным программированием (ООП), которое сосредоточено в первую очередь на данных и манипуляциях ими. Традиционное процедурное программирование, с другой стороны, следует парадигме *нисходящего программирования*, когда сначала разрабатывается укрупненная модульная структура, а затем внимание переключается на детали. Оба метода полезны и оба ведут к получению модульных программ.

Обычная идиома функций для обработки массивов

Предположим, что функция должна обрабатывать массив значений, скажем, типа `double`. Если функция предназначена для изменения массива, прототип может выглядеть так:

```
void f_modify(double ar[], int n);
```

Если функция сохраняет значения, прототип может выглядеть следующим образом:

```
void _f_no_change(const double ar[], int n);
```

Разумеется, имена переменных в прототипах могут быть опущены, а возвращаемый тип может отличаться от `void`. Основные моменты состоят в том, что `ar` в действительности является указателем на первый элемент переданного массива, а также в том, что количество элементов передается в качестве аргумента, поэтому функция

может использоваться с любым размером массива при условии, что он содержит значения типа double:

```
double rewards[1000];
double faults[50];
...
f_modify(rewards, 1000);
f_modify(faults, 50);
```

Эта идиома (передача имени массива и его размера в виде аргументов) работает за счет передачи двух чисел — адреса массива и количества элементов. Как уже было показано, функция теряет некоторую информацию об исходном массиве; например, использовать sizeof для получения размера в ней нельзя, поэтому нужно полагаться на то, что при вызове будет передано корректное количество элементов.

Функции, работающие с диапазонами массивов

Как вы уже видели, функции C++, обрабатывающие массивы, нуждаются в информации относительно типа данных, хранящихся в массиве, местоположения его начала и количества его элементов. Традиционный подход C/C++ к функциям, обрабатывающим массивы, состоит в передаче указателя на начало массива в одном аргументе и размера массива — в другом. (Указатель сообщает функции и то, где искать массив, и тип его элементов.) Это предоставляет функции исчерпывающую информацию, необходимую для нахождения данных.

Существует другой подход к предоставлению функции нужной информации — указание *диапазона* элементов. Это можно сделать, передав два указателя — один, идентифицирующий начальный элемент массива, и второй, указывающий его конец. Стандартная библиотека шаблонов C++ (STL; рассматривается в главе 16), например, обобщает такой подход с применением диапазона. Подход STL использует концепцию "следующий после конца" для указания границы диапазона. То есть в случае массива аргументом, идентифицирующим конец массива, должен быть указатель, который установлен на адрес, следующий сразу за последним элементом. Например, предположим, что имеется такое объявление:

```
double elbuod[20];
```

Диапазон определяют два указателя — elbuod и elbuod + 20. Первый — elbuod — это имя массива; он указывает на первый элемент. Выражение elbuod + 19 указывает на последний элемент (т.е. elbuod[19]), поэтому elbuod + 20 указывает на элемент, следующий сразу за последним. Передавая функции диапазон, вы сообщаете ей, какие элементы должны обрабатываться. В листинге 7.8 приведен измененный код из листинга 7.6, в котором используются два указателя для задания диапазона.

Листинг 7.8. arrfun4.cpp

```
// arrfun4.cpp -- функция с диапазоном массива
#include <iostream>
const int ArSize = 8;
int sum_arr(const int * begin, const int * end);
int main()
{
    using namespace std;
    int cookies[ArSize] = {1,2,4,8,16,32,64,128};
    // Некоторые системы требуют предварить int словом static,
    // чтобы разрешить инициализацию массива
```

```
    int sum = sum_arr(cookies, cookies + ArSize);
    cout << "Total cookies eaten: " << sum << endl;
    sum = sum_arr(cookies, cookies + 3);          // три первых элемента
    cout << "First three eaters ate " << sum << " cookies.\n";
    sum = sum_arr(cookies + 4, cookies + 8);      // четыре последних элемента
    cout << "Last four eaters ate " << sum << " cookies.\n";
    return 0;
}

// Возвращает сумму элементов целочисленного массива
int sum_arr(const int * begin, const int * end)
{
    const int * pt;
    int total = 0;
    for (pt = begin; pt != end; pt++)
        total = total + *pt;
    return total;
}
```

Ниже показан пример вывода программы из листинга 7.8:

```
Total cookies eaten: 255
First three eaters ate 7 cookies.
Last four eaters ate 240 cookies.
```

Замечания по программе

В листинге 7.8 обратите внимание на цикл внутри функции `sum_array()`:

```
for (pt = begin; pt != end; pt++)
    total = total + *pt;
```

Здесь указатель `pt` устанавливается на первый обрабатываемый элемент (на который указывает `begin`) и прибавляет `*pt` (значение самого элемента) к общей сумме `total`. Затем цикл обновляет `pt`, увеличивая его на единицу, после чего он указывает на следующий элемент. Процесс продолжается до тех пор, пока `pt != end`. Когда `pt`, наконец, становится равным `end`, он указывает на позицию, следующую за последним элементом диапазона, поэтому цикл завершается.

Второе, что следует отметить — это то, как разные вызовы функций задают различные диапазоны в пределах массива:

```
int sum = sum_arr(cookies, cookies + ArSize);
...
sum = sum_arr(cookies, cookies + 3);          // три первых элемента
...
sum = sum_arr(cookies + 4, cookies + 8);      // четыре последних элемента
```

Значение `cookies + ArSize` указывает на позицию, следующую за последним элементом массива. (Массив содержит `ArSize` элементов, потому `cookies[ArSize-1]` является последним элементом с адресом `cookies + ArSize - 1`.) Поэтому диапазон `cookies, cookies + ArSize` определяет весь массив. Аналогично `cookies, cookies + 3` означает первые три элемента и т.д.

Кстати, обратите внимание, что согласно правилам вычитания указателей, в `sum_arr()` выражение `end - begin` дает целочисленное значение, равное количеству элементов в диапазоне.

Кромс того, важно передавать указатели в корректном порядке; в коде предполагается, что `end` поступает после `begin`.

Указатели и const

Использование ключевого слова const с указателями характеризуется рядом тонких моментов (с указателями всегда связаны тонкие аспекты), поэтому присмотримся к нему повнимательнее. Применять ключевое слово const с указателями можно двумя разными способами. Первый — заставить указатель указывать на константный объект, тем самым предотвращая модификацию объекта через указатель. Второй способ — сделать сам указатель константным, запретив переустанавливать его на что-нибудь другое. Теперь обратимся к деталям.

Сначала объявим pt как указатель на константу:

```
int age = 39;
const int * pt = &age;
```

Это объявление устанавливает, что pt указывает на const int (в данном случае — 39). Таким образом, вы не сможете использовать pt для изменения этого значения. Другими словами, значение *pt является константным и не может быть изменено:

```
*pt += 1;        // НЕПРАВИЛЬНО, потому что pt указывает на const int
cin >> *pt;      // НЕПРАВИЛЬНО по той же причине
```

Теперь проанализируем тонкие моменты. Такое объявление pt не обязательно значит, что значение, на которое он указывает, действительно является константой; это значит лишь, что значение постоянно, только когда к нему обращаются через pt. Например, pt указывает на age, а age — не константа. Вы можете изменить значение age непосредственно, используя переменную age, но вы не можете изменить это значение через указатель pt:

```
*pt  =  2  0  ;// НЕПРАВИЛЬНО, потому что pt указывает на const int
age  =  2  0  ;// ПРАВИЛЬНО, потому что age не объявлено как const
```

В предыдущих примерах вы присваивали адрес обычной переменной обычному указателю. В этом примере вы присваиваете адрес обычной переменной указателю на константу. Это оставляет две другие возможности: присваивание адреса константной переменной указателю на константу и присваивание адреса константной переменной обычному указателю. Возможно ли то и другое? Первое — да, второе — нет:

```
const float g_earth = 9.80;
const float * pe = &g_earth;   // ПРАВИЛЬНО

const float g_moon = 1.63;
float * pm = &g_moon;          // НЕПРАВИЛЬНО
```

В первом случае вы не сможете использовать ни g_earth, ни pe для изменения значения 9.8. Второй случай в C++ не допускается по простой причине: если вы можете присвоить адрес g_moon указателю pm, то можете схитрить и применить pm для изменения g_moon. Это сводит на нет константный статус g_moon, поэтому C++ запрещает присваивание адреса константной переменной не константному указателю. (При крайней необходимости вы можете использовать приведение типа для преодоления подобного ограничения; см. в главе 15 обсуждение операции const_cast.)

Ситуация становится несколько более сложной, когда вы имеете дело с указателями на указатели. Как было показано ранее, присваивание не константного указателя константному разрешено, если вы имеете дело только с одним уровнем косвенности:

```
int age = 39;                  // age++ - допустимая операция
int * pd = &age;               // *pd = 41 - допустимая операция
const int * pt = pd;           // *pt = 42 - недопустимая операция
```

Однако присваивания указателей, которые смешивают константы и не константы в такой манере, становятся небезопасными, когда это делается на двух уровнях косвенности. Если смешивание констант и не констант было разрешено, вы могли бы написать что-нибудь вроде такого:

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;   // не разрешено, но предположим иначе
*pp2 = &n;   // правильно, оба const, но устанавливает в p1 указатель на n
*p1 = 10;    // правильно, но изменяет const n
```

Здесь код присваивает не константный адрес (`&p1`) константному указателю (`pp2`), что позволяет `p1` применяться для изменения константных данных. Таким образом, правило, гласящее, что вы можете присваивать не константный адрес или указатель константному указателю, работает только в том случае, когда есть лишь один уровень косвенности — например, если указатель указывает на базовый тип данных.

На заметку!

Вы можете присваивать адрес как константных, так и не константных данных указателю на константу, предполагая, что эти данные сами не являются указателем, но присвоить адрес не константных данных допускается только не константному указателю.

Предположим, что есть массив константных данных:

```
const int months[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

Запрет на присваивание адреса константного массива означает, что вы не можете передавать имя такого массива в качестве аргумента функции, используя не константный формальный аргумент:

```
int sum(int arr[], int n);     // должен быть const int arr[]
...
int j = sum(months, 12);       // не допускается
```

Этот вызов функции пытается присвоить константный указатель (`months`) не константному указателю (`arr`), и компилятор не разрешает такой вызов.

И пользуйте `const`, когда это возможно

Существуют две серьезных причины объявлять аргументы-указатели указателями на константные данные.

- Это защищает от программных ошибок, из-за которых могут непреднамеренно изменяться данные.
- Использование `const` позволяет функции обрабатывать как константные, так и не константные аргументы, в то время как функция, в прототипе которой `const` опущено, может принимать только не константные данные.

Вы должны объявлять формальные аргументы-указатели как указатели на `const`, где это только возможно.

Касательно еще одного тонкого момента рассмотрим следующее объявление:

```
int age = 39;
const int * pt = &age;
```

`const` во втором объявлении только предотвращает изменение того значения, на которое указывает `pt`, в данном случае `39`. Это не предотвращает изменение самого `pt`.

То есть вы вполне можете присвоить ему другой адрес:

```
int sage = 80;
pt = &sage;                    // может указывать на другое место
```

Но вы по-прежнему не сможете использовать pt для изменения того значения, на которое он указывает.

Второй способ применения const делает невозможным изменение самого указателя:

```
int sloth = 3;
const int * ps = &sloth;       // указатель на const int
int * const finger = &sloth;   // const-указатель на int
```

Обратите внимание, что в последнем объявлении позиция ключевого слова const изменена. Это объявление ограничивает finger тем, что он может указывать только на sloth и ни на что другое. Однако оно позволяет применить finger для изменения значения самого sloth. Второе из трех приведенных объявлений не разрешает применять ps для изменения значения sloth, но разрешает ps указывать на другое место памяти. Короче говоря, и finger и *ps являются константами, а *finger и ps — нет (рис. 7.5).

При желании можно объявить константный указатель на константный объект:

```
double trouble = 2.0E30;
const double * const stick = &trouble;
```

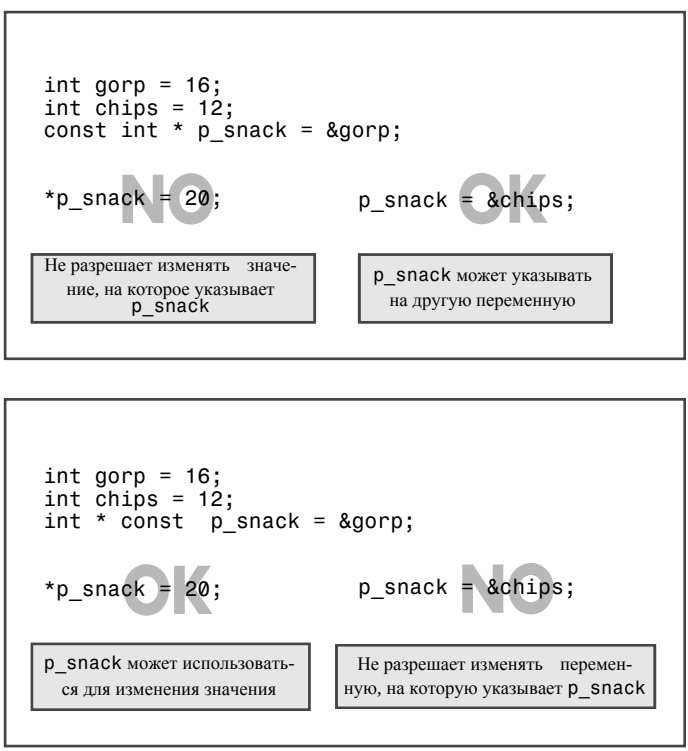


***Рис. 7.5.** Указатели на константы и константные указатели*

Здесь stick может указывать только на trouble, и stick не может применяться для изменения значения trouble. Короче говоря, и stick, и *stick являются const.

Обычно форма указателя на const используется для защиты данных при передаче указателя в качестве аргумента функции.

Например, вспомните прототип show_array() из листинга 7.5:

```
void show_array(const double ar[], int n);
```

Применение const в этом объявлении означает, что функция show_array() не может изменять значения в переданном ей массиве. Этот прием работает до тех пор, пока есть только один уровень косвенности. Здесь, например, элементы массива относятся к базовому типу, но если бы они были указателями или указателями на указатели, использовать const не удалось бы.

Функции и двумерные массивы

При написании функции, которая принимает в качестве аргумента двумерный массив, необходимо помнить, что имя массива трактуется как его адрес, поэтому соответствующий формальный параметр является указателем — так же, как и в случае одномерного массива. Сложность заключается в том, чтобы правильно объявить указатель. Предположим, например, что вы начинаете с такого кода:

```
int data[3][4] = {{1,2,3,4}, {9,8,7,6}, {2,4,6,8}};
int total = sum(data, 3);
```

Как должен выглядеть прототип sum()? И почему функция передаст количество строк (3), но не передает количество столбцов (4)?

Итак, data — имя массива из трех элементов. Первый элемент сам по себе является массивом их четырех значений типа int. То есть тип data — это указатель на массив из четырех int, поэтому соответствующий прототип должен быть таким:

```
int sum(int (*ar2)[4], int size);
```

Скобки необходимы, потому что показанное ниже объявление определило бы массив их четырех указателей на int вместо одного указателя на массив из четырех int, а параметр функции не может быть массивом:

```
int *ar2[4]
```

Существует альтернативный формат, который означает в точности то же самое, что и первый прототип, но, возможно, является более простым для чтения:

```
int sum(int ar2[][4], int size);
```

И тот, и другой прототип устанавливает, что ar2 — указатель, а не массив. Также обратите внимание, что тип указателя явно говорит о том, что он указывает на массив из четырех int. Таким образом, тип указателя задает количество столбцов — вот почему количество столбцов не передается в отдельном аргументе функции.

Поскольку тип указателя задает количество столбцов, функция sum() работает только с массивами из четырех столбцов. Однако количество строк задастся переменной size, поэтому sum() может работать с произвольным количеством строк:

```
int a[100][4];
int b[6][4];
...
```

```
int total1 = sum(a, 100);          // сумма всех элементов a
int total2 = sum(b, 6);            // сумма всех элементов b
int total3 = sum(a, 10);           // сумма первых 10 строк a
int total4 = sum(a+10, 20);        // сумма следующих 20 строк a
```

Зная, что ar2 — указатель на массив, как его можно использовать в определении функции? Простейший способ — работать с ar2 как с именем двумерного массива. Вот возможный вариант определения функции:

```
int sum(int ar2[][4], int size)
{
   int total = 0;
   for (int r = 0; r < size; r++)
      for (int c = 0; c < 4; c++)
         total += ar2[r][c];
   return total;
}
```

Еще раз обратите внимание, что количество строк передается в параметре size, но количество столбцов является фиксированным и равно 4, как в объявлении ar2, так и во вложенном цикле.

Вот почему можно использовать нотацию массивов. Поскольку ar2 указывает на первый элемент (элемент 0) массива, элементы которого являются массивами из четырех int, то выражение ar2+r указывает на элемент номер r. Таким образом, ar2[r] — это элемент номер r. Этот элемент сам по себе является массивом из четырех int, поэтому ar2[r] — имя этого массива из четырех int. Применение индекса к имени массива дает нам его элемент, поэтому ar2[r][c] — элемент массива из четырех int, т.е. отдельное значение типа int. Для получения данных указатель ar2 должен быть разыменован дважды. Простейший способ сделать это — дважды использовать квадратные скобки, как в ar2[r][c]. Если это неудобно, можно два раза применить операцию *:

```
ar2[r][c] == *(*(ar2 + r) + c)      // одно и то же
```

Чтобы понять это, понадобится разобрать выражение по частям, начав изнутри:

```
ar2                  // указатель на первую строку — массив из 4 int
ar2 + r              // указатель на строку r (массив из 4 int)
*(ar2 + r)           // строка r (массив из 4 int, следовательно, имя массива,
                     // таким образом, указатель на первый int в строке, т.е.
ar2[r])
*(ar2 +r) + c        // указатель на элемент int под номером c в строке r,
                     // т.е. ar2[r] + c
*(*(ar2 + r) + c     // значение int под номером c в строке r, т.е. ar2[r][c]
```

Кстати, в коде sum() не используется const в объявлении параметра ar2, потому что эта техника предназначена для указателей на базовые типы, а ar2 — это указатель на указатель.

Функции и строки в стиле C

Вспомните, что строка в стиле C состоит из последовательности символов, ограниченных нулевым символом. Большая часть того, что вы изучили о проектировании функций массивов, также касается функций, обрабатывающих строки. Например, передача строки как аргумента означает передачу ее адреса, и вы можете использовать const для защиты содержимого строки от нежелательных изменений. Однако существует ряд особенностей, связанных со строками, о которых мы поговорим сейчас.

Функции с аргументами — строками в стиле C

Предположим, что требуется передать строку функции в виде аргумента. Доступны три варианта представления строки:

- массив `char`;
- константная строка в двойных кавычках (также называемая *строковым литералом*);
- указатель на `char`, установлены в адрес начала строки.

Все три варианта, однако, являются типом указателя на `char` (или, короче, тип `char *`), поэтому все три можно использовать в качестве аргументов функций, обрабатывающих строки:

```
char ghost[15] = "galloping";
char * str = "galumphing";
int n1 = strlen(ghost);            // ghost - это &ghost[0]
int n2 = strlen(str);              // указатель на char
int n3 = strlen("gamboling");      // адрес строки
```

Неформально вы можете сказать, что передаете строку как аргумент, но на самом деле вы передаете адрес ее первого символа. Это подразумевает, что прототип строковой функции должен использовать `char *` как тип формального параметра, представляющего строку.

Одно важное отличие между строкой в стиле C и обычным массивом состоит в том, что строка имеет встроенный ограничивающий нулевой символ. (Вспомните, что массив `char`, который содержит символы, но не содержит нулевой символ — это просто массив, а не строка.) Это значит, что вы не должны передавать размер строки в качестве аргумента. Вместо этого функция может использовать цикл для поочередного чтения каждого символа строки до тех пор, пока не будет достигнут ограничивающий нулевой символ.

Код в листинге 7.9 иллюстрирует этот подход для функции, выполняющей подсчет количества появлений определенного символа в строке. Поскольку программа не нуждается в обработке отрицательных значений, в качестве типа счетчика применяется `unsigned int`.

Листинг 7.9. `strgfun.cpp`

```
// strgfun.cpp -- функция со строковым аргументом
#include <iostream>
unsigned int c_in_str(const char * str, char ch);
int main()
{
   using namespace std;
   char mmm[15] = "minimum";                          // строка в массиве
   // Некоторые системы требуют предварить char словом static,
   // чтобы разрешить инициализацию массива
   char *wail = "ululate";                            // wail указывает на строку
   unsigned int ms = c_in_str(mmm, 'm');
   unsigned int us = c_in_str(wail, 'u');
   cout << ms << " m characters in " << mmm << endl;  // вывод количества символов m
   cout << us << " u characters in " << wail << endl; // вывод количества символов u
   return 0;
}
```

```
// Эта функция подсчитывает количество символов ch в строке str
unsigned int c_in_str(const char * str, char ch)
{
   unsigned int count = 0;
   while (*str)          // завершение, когда *str равно '\0'
   {
      if (*str == ch)
         count++;
      str++;             // перемещение указателя на следующий символ
   }
   return count;
}
```

Ниже показан вывод программы из листинга 7.9:

```
3 m characters in minimum
2 u characters in ululate
```

Замечания по программе

Поскольку функция `c_int_str()` в листинге 7.9 не должна изменять исходную строку, она использует модификатор `const` в объявлении формального параметра `str`. После этого, если вы ошибочно позволите функции изменить часть строки, компилятор перехватит эту ошибку. Разумеется, для объявления `str` в заголовке функции можно применять нотацию массивов:

```
unsigned int c_in_str(const char str[], char ch)  // также нормально
```

Однако использование нотации указателей напоминает о том, что аргумент не должен быть именем массива, а какой-то другой формой указателя.

Сама функция демонстрирует стандартный способ обработки символов в строке:

```
while (*str)
{
   операторы
   str++;
}
```

Изначально `str` указывает на первый символ строки, поэтому `*str` представляет сам первый символ. Например, непосредственно после первого вызова функции `*str` имеет значение `m` — первый символ в `minimum`. До тех пор, пока символ не является нулевым (`\0`), `*str` не равно нулю и цикл продолжается. В конце каждого шага цикла выражение `str++` увеличивает указатель на 1 байт, так что он указывает на следующий символ в строке. В конечном итоге `str` указывает на завершающий нулевой символ, что делает `*str` равным нулю, и цикл прекращается.

Функции, возвращающие строки в стиле C

Теперь предположим, что требуется написать функцию, возвращающую строку. Конечно, функция может это сделать. Но она может вернуть адрес строки, и это наиболее эффективно. Например, листинг 7.10 определяет функцию `buildstr()`, возвращающую указатель. Эта функция принимает два аргумента: символ и число. Используя `new`, она создает строку, длина которой равна переданному числу, и инициализирует каждый ее элемент значением переданного символа. Затем она возвращает указатель на эту новую строку.

Листинг 7.10. strgback.cpp

```
// strgback.cpp -- функция, возвращающая указатель на char
#include <iostream>
char * buildstr(char c, int n);          // прототип
int main()
{
   using namespace std;
   int times;
   char ch;
   cout << "Enter a character: ";        // ввод символа
   cin >> ch;
   cout << "Enter an integer: ";         // ввод целого числа
   cin >> times;
   char *ps = buildstr(ch, times);
   cout << ps << endl;
   delete [] ps;                         // освобождение памяти
   ps = buildstr('+', 20);               // повторное использование указателя
   cout << ps << "-DONE-" << ps << endl;
   delete [] ps;                         // освобождение памяти
   return 0;
}

// Строит строку из n символов с
char * buildstr(char c, int n)
{
   char * pstr = new char[n + 1];
   pstr[n] = '\0';                       // завершение строки
   while (n-- > 0)
      pstr[n] = c;                       // заполнение остатка строки
   return pstr;
}
```

Ниже показан пример запуска программы из листинга 7.10:

```
Enter a character: V
Enter an integer: 46
VVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVVV
++++++++++++++++++-DONE-++++++++++++++++++
```

Замечания по программе

Чтобы создать строку из `n` видимых символов, понадобится разместить `n + 1` символов, с учетом нулевого символа-ограничителя. Поэтому функция в листинге 7.10 запрашивает `n + 1` байт для размещения строки. Далее она устанавливает последний байт в нулевой символ. После этого наполняет массив от конца к началу. Следующий цикл в листинге 7.10 выполняется `n` раз — уменьшая `n` до `0` и заполняя `n` элементов:

```
while (n-- > 0)
   pstr[n] = c;
```

Перед началом последнего прохода цикла `n` имеет значение `1`. Поскольку `n--` означает использовать значение, а потом инкрементировать его, условие цикла `while` проверяет `1` на равенство `0`, возвращает `true`, и цикл продолжается. Но после выполнения этой проверки функция уменьшает `n` до `0`, поэтому `pstr[0]` — последний элемент, которому присваивается `c`. Причина наполнения строки от конца к началу вместо того, чтоб наполнять от начала к концу, связана с желанием избежать применения дополнительной переменной.

Использование противоположного порядка потребовало бы примерно такого кода:

```
int i = 0;
while (i < n)
   pstr[i++] = c;
```

Обратите внимание, что переменная pstr является локальной по отношению к функции buildstr(), поэтому, когда эта функция завершается, память, занятая pstr (но не самой строкой), освобождается. Но поскольку функция возвращает значение pstr, программа имеет возможность получить доступ к новой строке через указатель ps в main().

Программа из листинга 7.10 применяет delete, чтобы освободить память, когда необходимость в строке отпадает. Затем она повторно использует ps, чтобы указать на новый блок памяти, полученный для следующей строки, и снова освобождает ее. Недостаток такого подхода (функция, возвращающая указатель на память, выделенную операцией new) состоит в том, что он возлагает на программиста ответственность за вызов delete. В главе 12 вы увидите, что классы C++ за счет использования конструкторов и деструкторов могут самостоятельно позаботиться об этих деталях.

Функции и структуры

Теперь давайте перейдем от массивов к структурам. Создавать функции для структур гораздо легче, чем для массивов. Хотя структурные переменные подобны массивам в том, что и те, и другие могут содержать несколько элементов данных, когда речь идет об их применении в функциях, структурные переменные ведут себя как базовые переменные, имеющие единственное значение. То есть, в отличие от массивов, структуры связывают свои значения в одиночную сущность, или объект данных, который будет трактоваться как единое целое. Вспомните, что одну структуру можно присваивать другой. Аналогично, можно передавать структуры по значению, как это делается с обычными переменными. В этом случае функция работает с копией исходной структуры. Здесь нет никаких трюков вроде того, что имя массива является указателем на его первый элемент. Имя структуры — это просто имя структуры, и если необходим ее адрес, то вы можете получить его с помощью операции &. (В языках C и C++ символ & применяется в качестве операции взятия адреса, но в C++ этот символ дополнительно используется для идентификации ссылочных переменных, как будет показано в главе 8.)

Самый прямой способ использования структуры в программе — это трактовать их так же, как обычные базовые типы — т.е. передавать в виде аргументов и если нужно, то использовать их в качестве возвращаемых значений. Однако существует один недостаток в передаче структур по значению. Если структура велика, то затраты, необходимые для создания копии структуры, могут значительно увеличить потребности в памяти и замедлить работу программы. По этой причине (и еще потому, что изначально язык C не позволял передавать структуры по значению) многие программисты на C предпочитают передавать адрес структуры и затем использовать указатель для доступа к ее содержимому. В C++ предлагается третья альтернатива, которая называется *передачей по ссылке* и обсуждается в главе 8. Давайте сейчас рассмотрим два первых варианта, начиная с передачи и возврата целых структур.

Передача и возврат структур

Передача структур по значению имеет наибольший смысл, когда структура относительно компактна, поэтому давайте рассмотрим несколько примеров, демонст-

рирующих такой подход. Первый пример имеет дело со временем путешествия (не путать с путешествиями во времени). Некоторые карты сообщают, что на проезд из Thunder Falls в Bingo City нужно потратить 3 часа 50 минут, а на проезд из Bingo City в Grotesquo — 1 час 25 минут. Для представления этих периодов времени можно использовать структуру, один член которой предназначен для представления часов, а другой — для минут. Сложение двух периодов будет не простым, потому что придется преобразовывать минуты в часы, когда их сумма превышает 60. Например, два периода времени на дорогу, приведенные выше, дают в сумме 4 часа 75 минут, что должно быть преобразовано в 5 часов 15 минут. Давайте разработаем структуру для представления значений времени, а затем функцию, которая будет принимать две таких структуры в виде аргументов и возвращать структуру, представляющую их сумму.

Определить структуру просто:

```
struct travel_time
{
    int hours;
    int mins;
};
```

Далее рассмотрим прототип для функции `sum()`, который вернет сумму двух таких структур. Возвращаемое значение должно иметь тип `travel_time`, как и два ее аргумента. Таким образом, прототип должен выглядеть следующим образом:

```
travel_time sum(travel_time t1, travel_time t2);
```

Чтобы сложить два периода времени, сначала необходимо сложить минуты. Целочисленное деление на 60 даст количество часов, в которые сложатся минуты, а операция модуля (`%`) даст оставшиеся минуты. В листинге 7.11 этот подход воплощен в функции `sum()`; еще одна функция, `show_time()`, служит для отображения содержимого структуры `travel_time`.

Листинг 7.11. `travel.cpp`

```
// travel.cpp -- использование структур с функциями
#include <iostream>
struct travel_time
{
    int hours;
    int mins;
};
const int Mins_per_hr = 60;
travel_time sum(travel_time t1, travel_time t2);
void show_time(travel_time t);

int main()
{
    using namespace std;
    travel_time day1 = {5, 45};                  // 5 часов 45 минут
    travel_time day2 = {4, 55};                  // 4 часов 55 минут
    travel_time trip = sum(day1, day2);
    cout << "Two-day total: ";                   // итог за два дня
    show_time(trip);
    travel_time day3= {4, 32};
    cout << "Three-day total: ";                 // итог за три дня
    show_time(sum(trip, day3));
    return 0;
}
```

```
travel_time sum(travel_time t1, travel_time t2)
{
   travel_time total;
   total.mins = (t1.mins + t2.mins) % Mins_per_hr;
   total.hours = t1.hours + t2.hours +
    (t1.mins + t2.mins) / Mins_per_hr;
   return total;
}

void show_time(travel time t)
{
   using namespace std;
   cout << t.hours << " hours, "
        << t.mins << " minutes\n";              // часов, минут
}
```

Здесь `travel_time` действует как имя обычного стандартного типа; его можно использовать для объявления переменных, типа возврата функций и типа аргументов функций. Поскольку такие переменные, как `total` и `t1`, являются структурами `travel_time`, вы можете применять к ним операцию точки, чтобы обращаться к членам. Обратите внимание, что поскольку функция `sum()` возвращает структуру `travel_time`, ее можно использовать в качестве аргумента функции `show_time()`. А так как функции C++ по умолчанию передают аргументы по значению, то вызов `show_time(sum(trip, day3))` сначала вычислит `sum(trip, day3)`, чтобы получить ее возвращаемое значение. Затем это значение (а не сама функция) передается `show_time()`.

Ниже показан вывод программы из листинга 7.11:

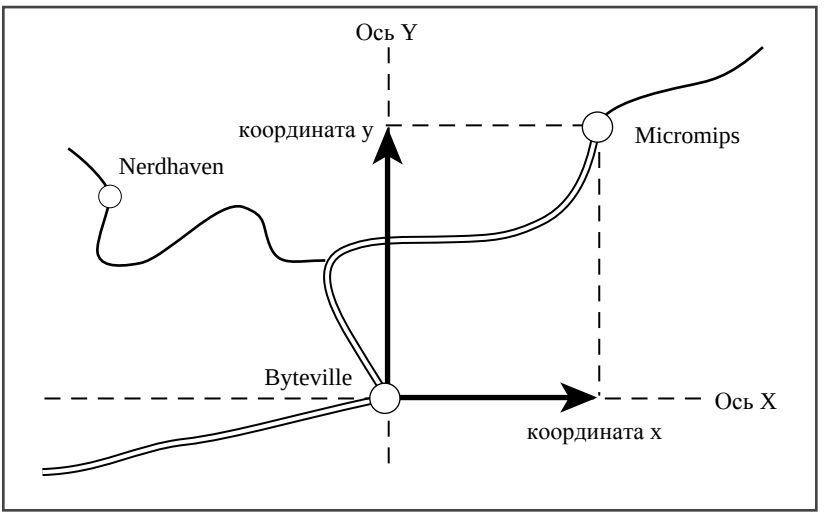
```
Two-day total: 10 hours, 40 minutes
Three-day total: 15 hours, 12 minutes
```

Еще один пример использования функций со структурами

Большая часть того, что вы узнали о функциях и структурах C++, в той же мере касается классов C++, поэтому имеет смысл рассмотреть второй пример. На этот раз мы будем иметь дело с пространством вместо времени. В частности, в этом примере мы определим две структуры, представляющие два разных способа описания координат на плоскости, и затем разработаем функции для преобразования одной формы в другую и отображения результата. В этом примере будет больше математики, чем в предыдущем, но вам не обязательно изучать математику, постигая C++.

Предположим, что вы хотите описать положение точки на экране или местоположение на карте относительно некоторой начальной точки. Одним из способов является отсчет горизонтального и вертикального смещений точки от начала координат. Традиционно в математике символ `x` используется для представления горизонтального смещения, а `y` — для вертикального (рис. 7.6). Вместе `x` и `y` формируют *прямоугольные координаты*. Для представления позиции можно определить структуру, состоящую из двух координат:

```
struct rect
{
   double x;    // расстояние по горизонтали от начальной точки
   double y;    // расстояние по вертикали от начальной точки
};
```

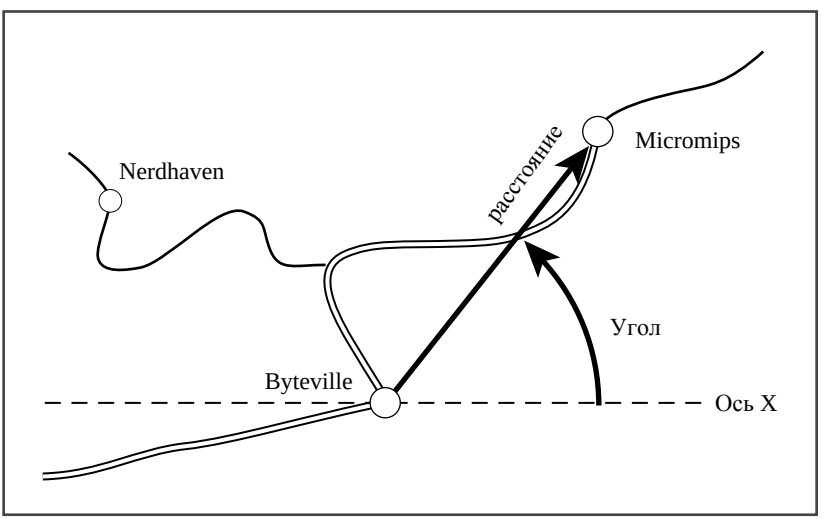


Прямоугольные координаты Micromips относительно Byteville

Рис. 7.6. Прямоугольные координаты

Второй способ описания позиции точки предусматривает указание ее расстояния от начала координат и направления (например, 40 градусов на север от восточного направления). Традиционно математики измеряют угол против часовой стрелки от положительной горизонтальной оси (рис. 7.7). Расстояние и угол вместе составляют *полярные координаты*. Для такого представления позиции можно определить вторую структуру:

```
struct polar
{
   double distance;      // расстояние от исходной точки
   double angle;         // направление
};
```



Полярные координаты Micromips относительно Byteville

Рис. 7.7. Полярные координаты

Теперь давайте построим функцию, которая будет отображать содержимое структуры типа polar. Математические функции в библиотеке C++ (позаимствованные из C) ожидают значения углов в радианах, поэтому мы тоже будем измерять углы в этих единицах. Но для целей отображения радианы могут быть преобразованы в градусы. Это означает умножение на $180/\pi$, что примерно составляет 57,29577951.

Вот эта функция:

```
// Отображение полярных координат с преобразованием радиан в градусы
void show_polar (polar dapos)
{
   using namespace std;
   const double Rad_to_deg = 57.29577951;

   cout << "distance = " << dapos.distance;
   cout << ", angle = " << dapos.angle * Rad_to_deg;
   cout << " degrees\n";
}
```

Обратите внимание, что формальный аргумент имеет тип polar. Когда вы передаете структуру polar этой функции, ее содержимое копируется в структуру dapos, и функция использует эту копию в своей работе. Поскольку dapos — структура, для идентификации ее членов в функции применяется операция членства (точка), которая рассматривалась в главе 4.

Теперь давайте попробуем написать функцию, которая преобразует прямоугольные координаты в полярные. Эта функция должна будет принимать в виде аргумента структуру rect и возвращать вызывающей функции структуру polar. Для выполнения необходимых вычислений понадобятся функции из библиотеки math, поэтому программа должна будет включить заголовочный файл cmath (math.h в более старых системах). Кроме того, в некоторых системах компилятору потребуется сообщить, что он должен загрузить библиотеку math (см. главу 1). Для получения расстояния на основе горизонтальной и вертикальной компонент можно воспользоваться теоремой Пифагора:

```
distance = sqrt( x * x + y * y)
```

Функция atan2() из библиотеки math вычисляет угол по значениям x и y:

```
angle = atan2(y, x)
```

(Доступна также функция atan(), но она не различает углы 180 градусов с разными знаками.)

С учетом этих формул, функцию преобразования координат можно записать следующим образом:

```
// Преобразование прямоугольных координат в полярные
polar rect_to_polar(rect xypos)    // тип polar
{
   polar answer;

   answer.distance =
      sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
   answer.angle = atan2(xypos.y, xypos.x);
   return answer;                  // возврат структуры polar
}
```

Теперь, когда функции готовы, написание остальной части программы не составляет особого труда. В листинге 7.12 показан окончательный код.

Листинг 7.12. strctfun.cpp

```
// strctfun.cpp -- функции с аргументами-структурами
#include <iostream>
#include <cmath>

// Объявления структур
struct polar
{
   double distance;      // расстояние от исходной точки
   double angle;         // направление от исходной точки
};
struct rect
{
   double x;             // расстояние по горизонтали от исходной точки
   double y;             // расстояние по вертикали от исходной точки
};

// Прототипы
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);

int main()
{
   using namespace std;
   rect rplace;
   polar pplace;
   cout << "Enter the x and y values: ";          // ввод значений x и y
   while (cin >> rplace.x >> rplace.y)           // ловкое использование cin
   {
      pplace = rect_to_polar(rplace);
      show_polar(pplace);
      cout << "Next two numbers (q to quit): ";
           // Ввод следующих двух чисел (q для завершения)
   }
   cout << "Done.\n";
   return 0;
}

// Преобразование прямоугольных координат в полярные
polar rect_to_polar(rect xypos)
{
   using namespace std;
   polar answer;
   answer.distance =
      sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
   answer.angle = atan2(xypos.y, xypos.x);
   return answer;                                // возврат структуры polar
}

// Отображение полярных координат с преобразованием радиан в градусы
void show_polar (polar dapos)
{
   using namespace std;
   const double Rad_to_deg = 57.29577951;
   cout << "distance = " << dapos.distance;
   cout << ", angle = " << dapos.angle * Rad_to_deg;
   cout << " degrees\n";
}
```

На заметку!

Некоторые компиляторы требуют явных инструкций для нахождения математической библиотеки. Например, для вызова старых версий g++ должна использоваться следующая командная строка:

```
g++ structfun.C -lm
```

Ниже показан пример выполнения программы из листинга 7.12:

```
Enter the x and y values: 30 40
distance = 50, angle = 53.1301 degrees
Next two numbers (q to quit): -100 100
distance = 141.421, angle = 135 degrees
Next two numbers (q to quit): q
Done.
```

Замечания по программе

Мы уже обсудили две функции из листинга 7.12, а теперь давайте разберемся, как программа использует `cin` для управления циклом `while`:

```
while (cin >> rplace.x >> rplace.y)
```

Вспомните, что `cin` — объект класса `istream`. Операция извлечения (>>) спроектирована так, что выражение `cin >> rplace.x` также является объектом этого типа. Как будет показано в главе 11, операции классов реализованы с помощью функций. Что в действительности происходит, когда используется `cin >> rplace.x`? Программа вызывает функцию, которая возвращает переменную типа `istream`. Если вы применяете операцию извлечения к объекту `cin >> rplace.x` (как в `cin >> rplace.x >> rplace.y`), то опять получаете объект класса `istream`. Таким образом, в конечном итоге проверочное условие цикла `while` вычисляется как `cin`, что, как вы помните, при использовании в контексте проверочного условия преобразуется в булевское значение `true` или `false`, в зависимости от того, успешным ли был ввод. Например, в цикле из листинга 7.12 `cin` ожидает от пользователя ввода двух чисел. Если же вместо этого пользователь вводит `q`, как показано в примере вывода программы, операция `cin >>` распознает, что `q` — не число. Она оставляет `q` во входной очереди и возвращает значение, преобразуемое в `false`, завершая выполнение цикла.

Сравните этот подход к чтению чисел со следующим более простым:

```
for (int i = 0; i < limit; i++)
{
   cout << "Enter value #" << (i + 1) << ": ";
   cin >> temp;
   if (temp < 0)
      break;
   ar[i] = temp;
}
```

Чтобы прекратить этот цикл до его завершения, вы вводите отрицательное число. Это ограничивает ввод только неотрицательными значениями. Такой подход может быть подходящим в некоторых программах, однако чаще в качестве условия прекращения цикла используется то, что не будет исключать определенные числовые значения из списка допустимых. Применение `cin >>` в качестве проверочного условия исключает это ограничение, поскольку обеспечивает прием любого допустимого числового ввода. Вам стоит запомнить этот трюк и использовать его всякий раз, когда нужно организовать в цикле ввод чисел. К тому же следует иметь в виду,

что нечисловой ввод выставляет условие ошибки, которое предотвращает чтение любого дальнейшего ввода. Если программа должна выполнять дальнейший ввод после завершения цикла, вы должны применять `cin.clear()`, чтобы сбросить состояние ошибки входного потока, и вдобавок прочитать ошибочный ввод, чтобы избавиться от него. Этот прием демонстрировался в листинге 7.7.

Передача адресов структур

Предположим, что вы хотите сэкономить время и пространство памяти за счет передачи адресов структуры вместо самой структуры. Для этого потребуется переписать функции так, чтобы они использовали в качестве аргументов указатели на структуры. Давайте посмотрим, как можно переписать функцию `show_polar()`. Для этого понадобится внести три изменения.

- При вызове функции передать ей адрес структуры (`&pplace`) вместо самой структуры (`pplace`).
- Определить формальный параметр как указатель на структуру `polar` — т.е. `polar *`. Поскольку функция не должна модифицировать структуру, дополнительно задать модификатор `const`.
- Поскольку формальный параметр теперь будет указателем на структуру вместо самой структуры, использовать операцию `->` вместо операции точки.

После внесения этих изменений функция будет выглядеть следующим образом:

```
// отображение полярных координат с преобразованием радиан в градусы
void show_polar (const polar * pda)
{
    using namespace std;
    const double Rad_to_deg = 57.29577951;
    cout << "distance = " << pda->distance;
    cout << ", angle = " << pda->angle * Rad_to_deg;
    cout << " degrees\n";
}
```

Теперь изменим `rect_to_polar()`. Это будет несколько сложнее, потому что исходная функция `rect_to_polar()` возвращает структуру. Чтобы воспользоваться всеми преимуществами эффективности указателей, придется также возвращать указатель вместо значения. Для этого необходимо передать функции два указателя на структуру. Первый будет указывать на преобразовываемую структуру, а второй — на структуру, содержащую результат преобразования. Вместо *возврата* новой структуры функция *модифицирует* структуру, существующую в вызывающей функции. Поэтому, хотя первый аргумент является константным указателей, второй аргумент — не `const`. Во всем остальном применимы те же принципы, что использовались для перевода `show_polar()` к аргументам-указателям. В листинге 7.13 показана переработанная программа.

Листинг 7.13. `strctptr.cpp`

```
// strctptr.cpp -- функции с аргументами-указателями на структуры
#include <iostream>
#include <cmath>

// Объявления структур
struct polar
{
```

```
   double distance;      // расстояние от исходной точки
   double angle;         // направление от исходной точки
};
struct rect
{
   double x;             // расстояние по горизонтали от исходной точки
   double y;             // расстояние по вертикали от исходной точки
};
// Прототипы
void rect_to_polar(const rect * pxy, polar * pda);
void show_polar (const polar * pda);
int main()
{
   using namespace std;
   rect rplace;
   polar pplace;
   cout << "Enter the x and y values: ";      // ввод значений x и y
   while (cin >> rplace.x >> rplace.y)
   {
      rect_to_polar(&rplace, &pplace);       // передача адресов
      show_polar(&pplace);                   // передача адресов
      cout << "Next two numbers (q to quit): ";
            // Ввод следующих двух чисел (q для завершения)
   }
   cout << "Done.\n";
   return 0;
}
// Отображение полярных координат с преобразованием  радиан в градусы
void show_polar (const polar * pda)
{
   using namespace std;
   const double Rad_to_deg = 57.29577951;
   cout << "distance = " << pda->distance;
   cout << ", angle = " << pda->angle * Rad_to_deg;
   cout << " degrees\n";
}
// Преобразование прямоугольных координат в полярные
void rect_to_polar(const rect * pxy, polar * pda)
{
   using namespace std;
   pda->distance =
      sqrt(pxy->x * pxy->x + pxy->y * pxy->y);
   pda->angle = atan2(pxy->y, pxy->x);
}
```

На заметку!

Некоторые компиляторы требуют явных инструкций для нахождения математической библиотеки. Например, для вызова старых версий g++ должна использоваться следующая командная строка:

```
g++ structfun.C -lm
```

С точки зрения пользователя программа из листинга 7.13 ведет себя точно так же, как программа из листинга 7.12. Скрытое отличие в том, что программа из листинга 7.12 работает с копиями структур, в то время как программа из листинга 7.13 использует указатели, позволяя функциям оперировать на исходных структурах.

Функции и объекты класса string

Хотя строки в стиле C и класс string служат в основном одним и тем же целям, класс string больше похож на структуру, чем на массив. Например, структуру можно присвоить другой структуре, а объект – другому объекту. Структуру можно передавать как единую сущность в функцию, и точно так же можно передавать объект. Когда требуется несколько строк, можно объявить одномерный массив объектов string вместо двумерного массива char.

В листинге 7.14 представлен короткий пример, в котором объявляется массив объектов string и передается функции, отображающей их содержимое.

Листинг 7.14. topfive.cpp

```
// topfive.cpp -- обработка массива объектов string
#include <iostream>
#include <string>
using namespace std;
const int SIZE = 5;
void display(const string sa[], int n);
int main()
{
    string list[SIZE];              // массив из 5 объектов string
    cout << "Enter your " << SIZE << " favorite astronomical sights:\n";
          // Ввод астрономических объектов
    for (int i = 0; i < SIZE; i++)
    {
        cout << i + 1 << ": ";
        getline(cin,list[i]);
    }
    cout << "Your list:\n";         // вывод списка астрономических объектов
    display(list, SIZE);
    return 0;
}
void display(const string sa[], int n)
{
    for (int i = 0; i < n; i++)
    cout << i + 1 << ": " << sa[i] << endl;
}
```

Ниже показан пример вывода программы из листинга 7.14:

```
Enter your 5 favorite astronomical sights:
1: Orion Nebula
2: M13
3: Saturn
4: Jupiter
5: Moon
Your list:
1: Orion Nebula
2: M13
3: Saturn
4: Jupiter
5: Moon
```

В этом примере важно отметить, что если не принимать во внимание функцию getline(), то эта программа воспринимает объекты string как любой встроенный

тип, подобный int. Если вам нужен массив string, вы просто используете обычный формат объявления массива:

```
string list[SIZE];                  // массив из 5 объектов string
```

Каждый элемент массива list — это объект string, и он может использоваться следующим образом:

```
getline(cin,list[i]);
```

Аналогично, формальный аргумент sa является указателем на объект string, поэтому sa[i] — объект типа string, и он может использоваться соответственно:

```
cout << i + 1 << ": " << sa[i] << endl;
```

Функции и объекты array

Объекты классов в C++ основаны на структурах, поэтому некоторые из соглашений, принятых для структур, применимы также и к классам. Например, функции можно передать объект по значению, и тогда она будет действовать на копии исходного объекта. В качестве альтернативы можно передать указатель на объект, что позволит функции оперировать на исходном объекте. Давайте рассмотрим пример использования шаблонного класса array из C++11.

Предположим, что имеется объект array, предназначенный для хранения расходов по четырем временам года:

```
std::array<double, 4> expenses;
```

(Вспомните, что использование класса array требует включения заголовочного файла array, а имя array является частью пространства имен std.) Если функция должна просто отобразить содержимое expenses, можно передать этот объект по значению:

```
show(expenses);
```

Но если функция должна модифицировать объект expenses, ей понадобится передать адрес этого объекта:

```
fill(&expenses);
```

(В следующей главе обсуждается альтернативный подход, предусматривающий применение ссылок.) Точно такой же подход использовался для структур в листинге 7.13. Как могут быть объявлены эти две функции?

Типом expenses является array<double, 4>, поэтому вот как должны выглядеть их прототипы:

```
void show(std::array<double, 4> da);   // da - объект
void fill(std::array<double, 4> * pa); // pa — указатель на объект
```

Приведенные выше соображения формируют основу примера программы. Дополнительно в программе реализовано несколько других возможностей. Во-первых, значение 4 заменяется символической константой:

```
const int Seasons = 4;
```

Во-вторых, добавляется константный объект array, содержащий четыре объекта string для представления времен года:

```
const std::array<std::string, Seasons> Snames =
   {"Spring", "Summer", "Fall", "Winter"};
```

Обратите внимание, что шаблон array не ограничен хранением базовых типов данных; он может также хранить типы классов. Полный код примера программы приведен в листинге 7.15.

Листинг 7.15. arrobj.cpp

```
//arrobj.cpp – функции с объектами array (C++11)
#include <iostream>
#include <array>
#include <string>
// Константные данные
const int Seasons = 4;
const std::array<std::string, Seasons> Snames =
   {"Spring", "Summer", "Fall", "Winter"};
// Функция для изменения объекта array
void fill(std::array<double, Seasons> * pa);
// Функция, использующая объект array, но не изменяющая его
void show(std::array<double, Seasons> da);
int main()
{
   std::array<double, Seasons> expenses;
   fill(&expenses);
   show(expenses);
   return 0;
}
void fill(std::array<double, Seasons> * pa)
{
  using namespace std;
  for (int i = 0; i < Seasons; i++)
  {
    cout << "Enter " << Snames[i] << " expenses: "; // ввод расходов по временам года
    cin >> (*pa)[i];
  }
}
void show(std::array<double, Seasons> da)
{
   using namespace std;
   double total = 0.0;
   cout << "\nEXPENSES\n";                          // вывод расходов по временам года
   for (int i = 0; i < Seasons; i++)
   {
      cout << Snames[i] << ": $" << da[i] << endl;
      total += da[i];
   }
   cout << "Total Expenses: $" << total << endl;   // вывод общей суммы расходов
}
```

Ниже показан пример запуска:

```
Enter Spring expenses: 212
Enter Summer expenses: 256
Enter Fall expenses: 208
Enter Winter expenses: 244
EXPENSES
Spring: $212
Summer: $256
Fall: $208
Winter: $244
Total: $920
```

Замечания по программе

Поскольку константный объект `array` по имени `Snames` объявлен перед всем функциями, он может применяться в любом следующем за ним объявлении функции. Подобно константе `Seasons`, объект `Snames` совместно используется во всем файле исходного кода. В программе отсутствует директива `using`, поэтому `array` и `string` должны применяться с квалификатором `str::`. Что излишне не усложнять программу и сконцентрировать внимание на том, как функции работают с объектами, никаких проверок допустимости вводимых пользователем данных в функции `fill()` не предпринимается. Функции `fill()` и `show()` имеют недостатки. Проблема, связанная с функцией `show()`, состоит в том, что `expenses` хранит четыре значения `double`, и создавать новый объект этого размера с последующим копированием в него значений `expenses` неэффективно. Эта проблема еще больше усугубится, если мы модифицируем программу для обработки расходов на ежемесячной или ежедневной основе и соответствующим образом расширим `expenses`.

Функция `fill()` избегает этой проблемы неэффективности за счет использования указателя, так что она оперирует на исходном объекте. Однако платой за это будет применение нотации, которая выглядит более сложной:

```
fill(&expenses);  // не забывайте о &
...
cin >> (*pa)[i];
```

В последнем операторе `pa` — это указатель на объект `array<double, 4>`, поэтому `*pa` является объектом, а `(*pa)[i]` — элементом в этом объекте. Круглые скобки необходимы для соблюдения приоритета операций. Логика очень проста, но в результате увеличиваются возможности допустить ошибку.

Как будет показано в главе 8, применение ссылок помогает решить и проблему эффективности, и проблему усложненной нотации.

Рекурсия

А теперь поговорим совершенно о другой теме. Функция C++ обладает интересной характеристикой — она может вызывать сама себя. (Однако, в отличие от C, в C++ функции `main()` не разрешено вызывать саму себя.) Эта возможность называется *рекурсией*. Рекурсия — важный инструмент в некоторых областях программированиях, таких как искусственный интеллект, но здесь мы дадим только поверхностные сведения о принципах ее работы.

Рекурсия с одиночным рекурсивным вызовом

Если рекурсивная функция вызывает саму себя, затем этот новый вызов снова вызывает себя и т.д., то получается бесконечная последовательность вызовов, если только код не включает в себе нечто, что позволит завершить эту цепочку вызовов. Обычный метод состоит в том, что рекурсивный вызов помещается внутрь оператора `if`. Например, рекурсивная функция типа `void` по имени `recurs()` может иметь следующую форму:

```
void recurs(списокАргументов)
{
    операторы1
    if (проверка)
        recurs(аргументы)
    операторы2
}
```

В какой-то ситуации *проверка* возвращает `false`, и цепочка вызовов прерывается.

Рекурсивные вызовы порождают замечательную цепочку событий. До тех пор, пока условие оператора `if` остается истинным, каждый вызов `recurs()` выполняет *операторы1* и затем вызывает новое воплощение `recurs()`, не достигая конструкции *операторы2*. Когда условие оператора `if` возвращает `false`, текущий вызов переходит к *операторы2*. Когда текущий вызов завершается, управление возвращается предыдущему экземпляру `recurs()`, который вызвал его. Затем этот экземпляр выполняет свой раздел *операторы2* и прекращается, возвращая управление предшествующему вызову, и т.д. Таким образом, если происходит пять вложенных вызовов `recurs()`, то первый раздел *операторы1* выполняется пять раз в том порядке, в котором произошли вызовы, а потом пять раз в обратном порядке выполняется раздел *операторы2*. После входа в пять уровней рекурсии программа должна пройти обратно эти же пять уровней. Код в листинге 7.16 демонстрирует описанное поведение.

Листинг 7.16. `recur.cpp`

```
// recur.cpp -- использование рекурсии
#include <iostream>
void countdown(int n);
int main()
{
   countdown(4);              // вызов рекурсивной функции
   return 0;
}
void countdown(int n)
{
   using namespace std;
   cout << "Counting down ... " << n << endl;
   if (n > 0)
      countdown(n-1);         // функция вызывает сама себя
   cout << n << ": Kaboom!\n";
}
```

Ниже приведен аннотированный вывод программы из листинга 7.16:

```
Counting down ... 4       ← уровень 1; добавление уровней рекурсии
Counting down ... 3       ← уровень 2
Counting down ... 2       ← уровень 3
Counting down ... 1       ← уровень 4
Counting down ... 0       ← уровень 5; финальный рекурсивный вызов
0: Kaboom!                ← уровень 5; начало обратного прохода
1: Kaboom!                ← уровень 4
2: Kaboom!                ← уровень 3
3: Kaboom!                ← уровень 2
4: Kaboom!                ← уровень 1
```

Обратите внимание, что каждый рекурсивный вызов создает собственный набор переменных, поэтому на момент пятого вызова она имеет пять отдельных переменных по имени `n` — каждая с собственным значением. Вы можете убедиться в этом, модифицировав код в листинге 7.16 таким образом, чтобы отображать адрес `n` наряду со значением:

```
cout << "Counting down ... " << n << " (n at " << &n << ")" << endl;
...
cout << n << ": Kaboom!"; << "          (n at " << &n << ")" << endl;
```

Если сделать так, то вывод программы примет следующий вид:

```
Counting down ... 4  (n at 0012FE0C)
Counting down ... 3  (n at 0012FD34)
Counting down ... 2  (n at 0012FC5C)
Counting down ... 1  (n at 0012FB84)
Counting down ... 0  (n at 0012FAAC)
0: Kaboom!           (n at 0012FAAC)
1: Kaboom!           (n at 0012FB84)
2: Kaboom!           (n at 0012FC5C)
3: Kaboom!           (n at 0012FD34)
4: Kaboom!           (n at 0012FE0C)
```

Как видите, переменная `n`, имеющая значение `4`, размещается в одном месте (в данном примере по адресу 0012FE0C), переменная `n` со значением `3` находится в другом месте (адрес памяти 0012FD34) и т.д. Кроме того, обратите внимание, что адрес переменной `n` для определенного уровня во время этапа `Counting down` совпадает с ее адресом для того же уровня во время этапа `Kaboom!`.

Рекурсия с множественными рекурсивными вызовами

Рекурсия, в частности, удобна в тех ситуациях, когда нужно вызывать повторяющееся разбиение задачи на две похожие подзадачи меньшего размера. Например, рассмотрим применение такого подхода для рисования линейки. Сначала нужно отметить два конца, найти середину и пометить ее. Затем необходимо применить ту же процедуру для левой половины линейки и правой ее половины. Если требуется больше частей, эта же процедура применяется для каждой из существующих частей. Такой рекурсивный подход иногда называют *стратегией "разделяй и властвуй"*. В листинге 7.17 данный подход иллюстрируется на примере рекурсивной функции `subdivide()`. Она использует строку, изначально заполненную пробелами, за исключением символов `|` на каждом конце. Затем главная программа запускает цикл из шести вызовов `subdivide()`, каждый раз увеличивая количество уровней рекурсии и печатая результирующую строку. Таким образом, каждая строка вывода представляет дополнительный уровень рекурсии. Чтобы напомнить о подобной возможности, вместо директивы `using` в программе применяется квалификатор `std::`.

Листинг 7.17. `ruler.cpp`

```
// ruler.cpp -- использование рекурсии для разделения линейки
#include <iostream>
const int Len = 66;
const int Divs = 6;
void subdivide(char ar[], int low, int high, int level);
int main()
{
   char ruler[Len];
   int i;
   for (i = 1; i < Len - 2; i++)
      ruler[i] = ' ';
   ruler[Len - 1] = '\0';
   int max = Len - 2;
   int min = 0;
   ruler[min] = ruler[max] = '|';
   std::cout << ruler << std::endl;
   for (i = 1; i <= Divs; i++)
   {
      subdivide(ruler,min,max, i);
```

```
        std::cout << ruler << std::endl;
        for (int j = 1; j < Len - 2; j++)
            ruler[j] = ' ';      // очистка линейки
    }
    return 0;
}
void subdivide(char ar[], int low, int high, int level)
{
    if (level == 0)
        return;
    int mid = (high + low) / 2;
    ar[mid] = '|';
    subdivide(ar, low, mid, level - 1);
    subdivide(ar, mid, high, level - 1);
}
```

Вот как выглядит вывод программы из листинга 7.17:

```
|                                                               |
|                               |                               |
|               |               |               |               |
|       |       |       |       |       |       |       |       |
|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
```

Замечания по программе

Функция `subdivide()` из листинга 7.17 использует переменную `level` для управления уровнем рекурсии. Когда эта функция вызывает саму себя, она уменьшает `level` на единицу, и как только `level` достигает нуля, функция завершается. Обратите внимание, что `subdivide()` вызывает себя дважды — один раз для левой части линейки и один — для правой. Исходная средняя точка становится правым концом для одного вызова и левым — для другого. Как видите, количество вызовов растет в геометрической прогрессии. То есть один вызов генерирует два, которые генерируют четыре вызова, те в свою очередь — восемь и т.д. Вот почему уровень 6 способен заполнить 64 элемента ($2^6 = 64$). Это непрерывное удвоение количества вызовов функции (а вместе с ними и количества сохраняемых переменных) делает такую форму рекурсии плохим решением при достаточно большом числе уровней. Если же уровней не слишком много, то это — простое и элегантное решение.

Указатели на функции

Любой разговор о функциях C и C++ будет неполным, если не упомянуть указатели на функции. Рассмотрим кратко эту тему, оставив более полное ее раскрытие специализированным источникам.

Функции, как и элементы данных, имеют адреса. Адрес функции — это адрес в памяти, где находится начало кода функции на машинном языке. Обычно пользователю ни к чему знать этот адрес, но это может быть полезно для программы. Например, можно написать функцию, которая принимает адрес другой функции в качестве аргумента. Это позволяет первой функции найти вторую и запустить ее. Такой подход сложнее, чем простой вызов второй функции из первой, но он открывает возможность передачи разных адресов функций в разные моменты времени. То есть первая функция может вызывать разные функции в разное время.

Основы указателей на функции

Проясним этот процесс на примере. Предположим, что требуется спроектировать функцию `estimate()`, которая оценивает затраты времени, необходимого для написания заданного количества строк кода, и вы хотите, чтобы этой функцией пользовались разные программисты. Часть кода `estimate()` будет одинакова для всех пользователей, но эта функция позволит каждому программисту применить собственный алгоритм оценки затрат времени. Механизм, используемый для обеспечения такой возможности, будет заключаться в передаче `estimate()` адреса конкретной функции, которая реализует алгоритм, выбранный данным программистом. Чтобы реализовать этот план, понадобится сделать следующее:

- получить адрес функции;
- объявить указатель на функцию;
- использовать указатель на функцию для ее вызова.

Получение адреса функции

Получить адрес функции очень просто: вы просто используете имя функции без скобок. То есть, если имеется функция `think()`, то ее адрес записывается как `think`. Чтобы передать функцию в качестве аргумента, вы просто передаете ее имя. Удостоверьтесь в том, что понимаете разницу между *адресом* функции и передачей ее *возвращаемого значения*:

```
process(think);  // передача адреса think() функции process()
thought(think()); // передача возвращаемого значения think() функции thought()
```

Вызов `process()` позволяет внутри этой функции вызвать функцию `think()`. Вызов `thought()` сначала вызывает функцию `think()` и затем передает возвращаемое ею значение функции `thought()`.

Объявление указателя на функцию

Чтобы объявить указатель на тип данных, нужно явно задать тип, на который будет указывать этот указатель. Аналогично, указатель на функцию должен определять, на функцию какого типа он будет указывать. Это значит, что объявление должно идентифицировать тип возврата функции и ее сигнатуру (список аргументов). То есть объявление должно предоставлять ту же информацию о функции, которую предоставляет и ее прототип. Например, предположим, что одна из функций для оценки затрат времени имеет следующий прототип:

```
double pam(int);     // прототип
```

Вот как должно выглядеть объявление соответствующего типа указателя:

```
double (*pf)(int);   // pf указывает на функцию, которая принимает
                     // один аргумент типа int и возвращает тип double
```

Совет

В общем случае для объявления указателя на функцию определенного рода можно сначала написать прототип обычной функции требуемого вида, а затем заменить ее имя выражением в форме `(*pf)`. В этом случае `pf` является указателем на функцию этого типа.

Объявление требует скобок вокруг `*pf`, чтобы обеспечить правильный приоритет операций. Скобки имеют более высокий приоритет, чем операция `*`, поэтому

*pf(int) означает, что pf() — функция, которая возвращает указатель, в то время как (*pf)(int) означает, что pf — указатель на функцию:

```
double (*pf)(int);   // pf указывает на функцию, возвращающую double
double *pf(int);     // pf() — функция, возвращающая указатель на double
```

После соответствующего объявления указателя pf ему можно присваивать адрес подходящей функции:

```
double pam(int);
double (*pf)(int);
pf = pam;            // pf теперь указывает на функцию pam()
```

Обратите внимание, что функция pam() должна соответствовать pf как по типу возврата, так и по сигнатуре. Компилятор отклонит несоответствующие присваивания:

```
double ned(double);
int ted(int);
double (*pf)(int);
pf = ned;            // неверно — несоответствие сигнатуры
pf = ted;            // неверно — несоответствие типа возврата
```

Вернемся к упомянутой ранее функции estimate(). Предположим, что вы хотите передавать ей количество строк кода, которые нужно написать, и адрес алгоритма оценки — функции, подобной pam(). Тогда она должна иметь следующий прототип:

```
void estimate(int lines, double (*pf)(int));
```

Это объявление сообщает, что второй аргумент является указателем на функцию, принимающую аргумент int и возвращающую значение double. Чтобы заставить estimate() использовать функцию pam(), вы передаете ей адрес pam:

```
estimate(50, pam);   // вызов сообщает estimate(),
                     // что она должна использовать pam()
```

Очевидно, что вся сложность использования указателей на функцию заключается в написании прототипов, в то время как передавать адрес очень просто.

Использование указателя для вызова функции

Теперь обратимся к завершающей части этого подхода — использованию указателя для вызова указываемой им функции. Ключ к этому находится в объявлении указателя. Вспомним, что там (*pf) играет ту же роль, что имя функции. Поэтому все, что потребуется сделать — использовать (*pf), как если бы это было имя функции:

```
double pam(int);
double (*pf)(int);
pf = pam;            // pf теперь указывает на функцию pam()
double x = pam(4);   // вызвать pam(), используя ее имя
double y = (*pf)(5); // вызвать pam(), используя указатель pf
```

В действительности C++ позволяет использовать pf, как если бы это было имя функции:

```
double y = pf(5);    // также вызывает pam(), используя указатель pf
```

Первая форма вызова более неуклюжа, чем эта, но она напоминает о том, что код использует указатель на функцию.

История против логики

О, великий синтаксис! Как `pf` и `(*pf)` могут быть эквивалентными? Сторонники одной школы утверждают, что поскольку `pf` — указатель на функцию, то `*pf` — функция, поэтому вы должны использовать для ее вызова `(*pf)()`. Сторонники другой школы придерживаются мнения, что поскольку имя функции является указателем на эту функцию, то и любой указатель на функцию должен вести себя как имя функции; отсюда вызов функции через указатель следует записывать как `pf()`. Язык C++ придерживается компромиссной точки зрения о том, что обе формы корректны, или, по крайней мере, допустимы, даже несмотря на то, что они логически несовместимы. Прежде чем вы отвергнете компромисс и выберете для себя одну форму, вспомните, что допущение несогласованных и логически несовместимых представлений вполне присуще человеческому мышлению.

Пример с указателем на функцию

В листинге 7.18 демонстрируется использование указателя функции в программе. Функция `estimate()` вызывается дважды — один раз с передачей ей адреса функции `betsy()`, а второй — адреса функции `pam()`. В первом случае `estimate()` применяет `betsy()` для вычисления необходимого количества часов, а во втором — использует для этого же `pam()`. Такое решение упростит развитие программы в будущем. Когда другой программист разработает собственный алгоритм оценки затрат времени, ему не придется переписывать `estimate()`. Вместо этого он должен просто реализовать свою функцию `ralph()`, обеспечив для нее необходимую сигнатуру и тип возврата. Конечно, переписать `estimate()` не трудно, но те же принципы применимы и к более сложному коду. К тому же метод указателей на функции позволяет модифицировать поведение `estimate()`, даже не имея доступа к ее исходному коду.

Листинг 7.18. `fun_ptr.cpp`

```
// fun_ptr.cpp -- указатели на функции
#include <iostream>
double betsy(int);
double pam(int);

// Второй аргумент – указатель на функцию double,
// которая принимает аргумент типа int
void estimate(int lines, double (*pf)(int));
int main()
{
   using namespace std;
   int code;
   cout << "How many lines of code do you need? "; // ввод количества строк кода
   cin >> code;
   cout << "Here's Betsy's estimate:\n";          // вывод первой оценки
   estimate(code, betsy);
   cout << "Here's Pam's estimate:\n";            // вывод второй оценки
   estimate(code, pam);
   return 0;
}
double betsy(int lns)
{
   return 0.05 * lns;
}
double pam(int lns)
{
   return 0.03 * lns + 0.0004 * lns * lns;
}
```

```
void estimate(int lines, double (*pf)(int))
{
    using namespace std;
    cout << lines << " lines will take ";
    cout << (*pf)(lines) << " hour(s)\n";       // вывод затрат времени
}
```

Ниже показан один из примеров выполнения программы из листинга 7.18:

```
How many lines of code do you need? 30
Here's Betsy's estimate:
30 lines will take 1.5 hour(s)
Here's Pam's estimate:
30 lines will take 1.26 hour(s)
```

А вот второй пример запуска той же программы:

```
How many lines of code do you need? 100
Here's Betsy's estimate:
100 lines will take 5 hour(s)
Here's Pam's estimate:
100 lines will take 7 hour(s)
```

Вариации на тему указателей на функции

Код с указателями на функции может выглядеть несколько устрашающе. Давайте рассмотрим пример, который иллюстрирует ряд проблем, связанных с указателями на функции, и демонстрирует способы их решения. Для начала ниже приведены прототипы некоторых функций, разделяющих одну и ту же сигнатуру и тип возврата:

```
const double * f1(const double ar[], int n);
const double * f2(const double [], int);
const double * f3(const double *, int);
```

Сигнатуры могут казаться разными, но все они одинаковы. Во-первых, вспомните, что в списке параметров для прототипа функции записи `const double ar[]` и `const double * ar` означают в точности одно и то же. Во-вторых, как вам уже должно быть известно, в прототипе можно опускать идентификаторы.

Следовательно, `const double ar[]` может быть сокращено до `const double []`, а `const double * ar` — до `const double *`. Таким образом, все показанные выше сигнатуры функций означают одно и то же. С другой стороны, определения функций предоставляют идентификаторы, поэтому ими будут либо `const double ar[]`, либо `const double * ar`.

Далее предположим, что необходимо объявить указатель, который мог бы указывать на одну из этих трех функций. Как уже было показано, подход заключается в том, что если `pa` является требуемым указателем, нужно взять прототип целевой функции и заменить в нем имя функции записью `(*pa)`:

```
const double * (*p1)(const double *, int);
```

Это можно скомбинировать с инициализацией:

```
const double * (*p1)(const double *, int) = f1;
```

Благодаря средству автоматического выведения типов C++11, объявление можно слегка упростить:

```
auto p2 = f2; // автоматическое выведение типа C++11
```

Теперь предположим, что есть следующие операторы:

```
cout << (*p1)(av,3) << ": " << *(*p1)(av,3) << endl;
cout << p2(av,3) << ": " << *p2(av,3) << endl;
```

Вспомните, что (*p1)(av,3) и p2(av,3) представляют вызов указываемой функции (в данном случае f1() и f2()) с аргументами av и 3. Следовательно, приведенные выше операторы должны вывести возвращаемые значения этих двух функций. Возвращаемые значения имеют тип const double * (т.е. адрес значения double). Таким образом, первая часть каждого выражения cout должна выводить адрес значения double. Чтобы увидеть действительно значение, хранящееся по этому адресу, к адресу необходимо применить операцию *, и именно это сделано в выражениях *(*p1)(av,3) и *p2(av,3).

При наличии трех функций для работы было бы удобно иметь массив указателей на функции. Затем его можно было использовать в цикле for для поочередного вызова каждой функции через ее указатель. Как это может выглядеть? Очевидно, что это должно в чем-то напоминать объявление одиночного указателя, но где-то должна присутствовать конструкция [3], которая отразит тот факт, что объявляется массив из трех указателей. Остается вопрос: где? А вот и ответ (включая инициализацию):

```
const double * (*pa[3])(const double *, int) = {f1,f2,f3};
```

Почему [3] находится именно в этом месте? Поскольку pa — это массив из трех элементов, началом объявления этого массива является pa[3]. Оставшаяся часть объявления относится к тому, что конкретно будет помещено в этот массив. Приоритет операции [] выше, чем *, поэтому *pa[3] говорит о том, что pa представляет собой массив из трех указателей. Остаток объявления отражает то, на что указывает каждый указатель: функция с сигнатурой const double *, int и типом возврата const double *. Следовательно, pa — это массив из трех указателей, каждый из которых указывает на функцию, принимающую const double * и int в качестве аргументов и возвращающую const double *.

Можно ли здесь воспользоваться ключевым словом auto? Нет, нельзя. Автоматическое выведение типа работает с одиночным инициализатором, но не со списком инициализации. Однако теперь, когда имеется массив pa, объявить указатель соответствующего типа очень легко:

```
auto pb = pa;
```

Как вы помните, имя массива — это указатель на его первый элемент, поэтому pa и pb являются указателями на указатель на функцию.

Как с их помощью вызвать функцию? И pa[i], и pb[i] представляют указатели в массиве, поэтому любой из них можно использовать для вызова функции следующим образом:

```
const double * px = pa[0](av,3);
const double * py = (*pb[1])(av,3);
```

Применив операцию *, можно получить значение double, на которое указывает указатель:

```
double x = *pa[0](av,3);
double y = *(*pb[1])(av,3);
```

Можно также сделать кое-что еще — создать указатель на целый массив. Так как имя массива pa уже является указателем на указатель на функцию, указатель на массив будет указателем на указатель на указатель. Это звучит несколько устрашающе,

но поскольку результат может быть представлен одиночным значением, допускается использование auto:

```
auto pc = &pa;        // автоматическое выведение типа C++11
```

А что, если вы предпочитаете это делать самостоятельно? Очевидно, что объявление должно быть похоже на объявление pa, но поскольку здесь присутствует дополнительный уровень косвенности, где-то понадобится еще одна операция *. В частности, если назвать новый указатель pd, то необходимо отразить, что он является указателем, а не именем массива. Это значит, что ядром объявления должно быть (*pd)[3]. Круглые скобки связывают идентификатор pd и *:

```
*pd[3]              // массив из трех указателей
(*pd)[3]            // указатель на массив из трех элементов
```

Другими словами, pd — это указатель, и он указывает на массив из трех элементов. Что это за элементы — описано в оставшейся части объявления pa. В результате этого подхода получаем следующее:

```
const double *(*(*pd)[3])(const double *, int) = &pa;
```

При написании вызова функции необходимо понимать, что если pd указывает на массив, то *pd — это сам массив, а (*pd)[i] — элемент массива, который представляет собой указатель на функцию. Таким образом, простейшая нотация для вызова функции выглядит как (*pd)[i](av,3), а *(*pd)[i](av,3) будет значением, на которое указывает возвращенный указатель. В качестве альтернативы можно было бы использовать второй синтаксис для обращения к функции через указатель: (*(*pd)[i])(av,3) для собственно вызова и *(*(*pd)[i])(av,3) для указываемого значения double.

Удостоверьтесь, что понимаете разницу между pa, которое представляет собой имя массива и является адресом, и &pa. Как было показано ранее, в большинстве контекстов pa является адресом первого элемента массива — т.е. &pa[0]. Следовательно, это адрес одиночного указателя. Но &pa представляет собой адрес всего массива (т.е. блока из трех указателей). С числовой точки зрения pa и &pa могут иметь одно и то же значение, но они относятся к разным типам. Одна практическая разница состоит в том, что pa+1 — это адрес следующего элемента в массиве, тогда как &pa+1 — адрес следующего за массивом pa блока из 12 байт (предполагается, что адреса имеют длину 4 байта). Другим отличием является то, что для получения значения первого элемента pa разыменовывается один раз, а &pa — два раза:

```
**&pa == *pa == pa[0]
```

Все, что обсуждалось выше, воплощено в листинге 7.19. Для целей иллюстрации функции f1() и т.д. сохранены предельно простыми. В программе в виде комментариев показаны альтернативы использованию auto, доступные в C++98.

Листинг 7.19 arfupt.cpp

```
// arfupt.cpp — массив указателей на функции
#include <iostream>
// Различные нотации, одни и те же сигнатуры
const double * f1(const double ar[], int n);
const double * f2(const double [], int);
const double * f3(const double *, int);
int main()
{
   using namespace std;
   double av[3] = {1112.3, 1542.6, 2227.9};
```

```
    // Указатель на функцию
    const double *(*p1)(const double *, int) = f1;
    auto p2 = f2;                              // автоматическое выведение типа C++11
    // До C++11 можно использовать следующий код
    // const double *(*p2)(const double *, int) = f2;
    // Использование указателей на функции
    cout << "Using pointers to functions:\n";
    cout << " Address Value\n";                // вывод значения адреса
    cout << (*p1)(av,3) << ": " << *(*p1)(av,3) << endl;
    cout << p2(av,3) << ": " << *p2(av,3) << endl;
    // pa – массив указателей
    // auto не работает со списковой инициализацией
    const double *(*pa[3])(const double *, int) = {f1,f2,f3};
    // но работает с инициализацией единственным значением
    // pb – указатель на первый элемент pa
    auto pb = pa;
    // До C++11 можно использовать следующий код
    // const double *(**pb)(const double *, int) = pa;
    // Использование массивов указателей на функции
    cout << "\nUsing an array of pointers to functions:\n";
    cout << " Address Value\n";                // вывод значения адреса
    for (int i = 0; i < 3; i++)
        cout << pa[i](av,3) << ": " << *pa[i](av,3) << endl;
    // Использование указателя на указатель на функцию
    cout << "\nUsing a pointer to a pointer to a function:\n";
    cout << " Address Value\n";                // вывод значения адреса
    for (int i = 0; i < 3; i++)
        cout << pb[i](av,3) << ": " << *pb[i](av,3) << endl;
    // Указатель на массив указателей на функции
    cout << "\nUsing pointers to an array of pointers:\n";
    cout << " Address Value\n";                // вывод значения адреса
    // Простой способ объявления pc
    auto pc = &pa;
    // До C++11 можно использовать следующий код
    // const double *(*(*pc)[3])(const double *, int) = &pa;
    cout << (*pc)[0](av,3) << ": " << *(*pc)[0](av,3) << endl;
    // Сложный способ объявления pd
    const double *(*(*pd)[3])(const double *, int) = &pa;
    // Сохранение возвращенного значения в pdb
    const double * pdb = (*pd)[1](av,3);
    cout << pdb << ": " << *pdb << endl;
    // Альтернативная нотация
    cout << (*(*pd)[2])(av,3) << ": " << *(*(*pd)[2])(av,3) << endl;
    // cin.get();
    return 0;
}
// Простейшие функции
const double * f1(const double * ar, int n)
{
    return ar;

const double * f2(const double ar[], int n)
{
    return ar+1;
}
const double * f3(const double ar[], int n)
{
    return ar+2;
}
```

Ниже показан вывод программы из листинга 7.19:

```
Using pointers to functions:
 Address    Value
002AF9E0:  1112.3
002AF9E8:  1542.6

Using an array of pointers to functions:
 Address    Value
002AF9E0:  1112.3
002AF9E8:  1542.6
002AF9F0:  2227.9

Using a pointer to a pointer to a function:
 Address    Value
002AF9E0:  1112.3
002AF9E8:  1542.6
002AF9F0:  2227.9

Using pointers to an array of pointers:
 Address    Value
002AF9E0:  1112.3
002AF9E8:  1542.6
002AF9F0:  2227.9
```

Показанные адреса являются местоположениями значений `double` в массиве `av`.

Этот пример может показаться несколько надуманным, однако указатели на массивы указателей на функции не является чем-то экзотическим. На самом деле такой подход используется в обычной реализации виртуальных методов класса (см. главу 13). К счастью, обо всех необходимых деталях заботится компилятор.

Высокая оценка `auto`

Одна из целей C++11 связана с упрощением использования языка C++, когда программист больше сконцентрирован на проектировании и меньше — на деталях. Код в листинге 7.19, безусловно, подтверждает это:

```
auto pc = &pa;                  // автоматическое выведение типа C++11
const double *(*(*pd)[3])(const double *, int) = &pa; // C++98; делается вручную
```

Средство автоматического выведения типов отражает философский сдвиг относительно роли компилятора. В C++98 компилятор использует свои знания, чтобы сообщить нам, когда мы допустили ошибку. В C++11, по крайней мере, с помощью средства `auto`, компилятор использует свои знания, чтобы помочь нам получить правильное объявление.

Здесь существует один потенциальный недостаток. Автоматическое выведение типов гарантирует, что тип переменной соответствует типу инициализатора, но по-прежнему существует возможность предоставления неверного типа инициализатора:

```
auto pc = *pa;                  // ошибка; вместо &pa указано *pa
```

Это объявление делает так, что тип `pc` совпадает с типом `*pa`, и приведет в итоге к ошибке компиляции, когда позже будет использоваться `pc`, исходя из предположения, что он имеет тот же самый тип, что и `&pa`.

Упрощение объявлений с помощью `typedef`

Помимо `auto`, в C++ предоставляются и другие инструменты, позволяющие упростить объявления. Вспомните из главы 5, что ключевое слово `typedef` позволяет создавать псевдоним типа:

```
typedef double real;      // делает real другим именем для типа double
```

Подход состоит в том, чтобы объявить псевдоним, как если бы он был идентификатором, и вставить перед ним ключевое слово typedef. Таким образом, чтобы сделать p_fun псевдонимом для типа указателя на функцию, используемого в листинге 7.19, потребуется записать следующий код:

```
typedef const double *(*p_fun)(const double *, int); // p_fun теперь
                                                      // является именем типа
p_fun p1 = f1;              // p1 указывает на функцию f1()
```

Затем это тип можно применять так, как показано ниже:

```
p_fun pa[3] = {f1,f2,f3}; // pa — массив из 3 указателей на функции
p_fun (*pd)[3] = &pa;     // pd указывает на массив из 3 указателей на функции
```

Использование typedef не только сокращает клавиатурный набор, но также уменьшает количество ошибок, допускаемых во время написания кода, и упрощает понимание программ.

Резюме

Функции — это программные модули C++. Чтобы использовать функцию, вы должны предоставить ее определение и прототип, после чего ее можно вызывать. Определение функции — это код, который реализует то, что она делает. Прототип функции описывает ее интерфейс: сколько она принимает параметров, какого типа эти параметры, и как выглядит тип возвращаемого ею значения, если оно есть. Вызов функции позволяет программе послать ей аргументы и передать поток управления коду функции.

По умолчанию функции C++ принимают аргументы по значению. Это значит, что формальные параметры в определении функции — это совершенно новые переменные, которые инициализируются значениями, переданными в вызове этой функции. Таким образом, C++ защищает целостность исходных данных, работая с копиями.

C++ трактует аргумент, являющийся именем массива, как адрес первого его элемента. Формально это по-прежнему означает передачу по значению, поскольку аргумент-указатель является копией исходного адреса, но функция использует его для обращения к содержимому исходного массива. Когда вы объявляете формальные параметры функции (и только в этом случае), следующие два объявления эквивалентны:

```
имяТипа arr[]
имяТипа * arr
```

Оба они означают, что arr — указатель на *имяТипа*. Однако при написании кода функции вы можете использовать arr для доступа к его элементам, как если бы он был именем массива: arr[i]. Даже при передаче указателей можно предохранить целостность исходных данных, объявив формальный аргумент как указатель на тип const. Поскольку передача адреса массива не сопровождается информацией о его размере, обычно размер массива передается в отдельном аргументе. В качестве альтернативы можно передавать указатель на начало массива и указатель на позицию, следующую сразу за последним элементом для определения диапазона, как это делается в алгоритмах в STL.

В C++ предусмотрено три способа представления строк в стиле C: символьный массив, строковая константа и указатель на строку. Все они имеют тип char* (указатель на символ), поэтому передаются функциям как аргумент типа char*. В C++ в качестве ограничителя строки используется нулевой символ (\0), и строковые функции выполняют проверку на нулевой символ для определения конца любой обрабатываемой строки.

В C++ также определен класс `string` для представления строк. Функции могут принимать объекты `string` в аргументах и использовать объекты `string` как возвращаемые значения. Метод `size()` класса `string` может применяться для определения длины хранимой в нем строки.

C++ трактует структуры точно так же, как базовые типы, в том смысле, что вы можете передавать их по значению и использовать в качестве типа возврата. Однако если структура достаточно велика, может оказаться более эффективным передавать указатель на структуру и позволить функции работать с исходными данными. Те же соображения применимы к объектам классов.

Функции C++ могут быть рекурсивными, т.е. код определенной функции может включать в себя вызов ее самой.

Имя функции C++ действует как ее адрес. Используя в функциях аргументы типа указателей на функции, вы можете передавать одной функции имя второй функции, если хотите, чтобы первая функция вызвала вторую.

Вопросы для самоконтроля

1. Назовите три шага по созданию функции.
2. Постройте прототипы, которые соответствовали бы следующим описаниям.
   а. `igor()` не принимает аргументов и не возвращает значения.
   б. `tofu()` принимает аргумент `int` и возвращает `float`.
   в. `mpg()` принимает два аргумента типа `double` и возвращает `double`.
   г. `summation()` принимает имя массива `long` и его размер и возвращает значение `long`.
   д. `doctor()` принимает строковый аргумент (строка не должна изменяться) и возвращает `double`.
   е. `ofcourse()` принимает структуру `boss` в качестве аргумента и не возвращает ничего.
   ж. `plot()` принимает указатель на структуру `map` в качестве аргумента и возвращает строку.
3. Напишите функцию, принимающую три аргумента: имя массива `int`, его размер и значение `int`. Функция должна присвоить каждому элементу массива это значение `int`.
4. Напишите функцию, принимающую три аргумента: указатель на первый элемент диапазона в массиве, указатель на элемент, следующий за концом этого диапазона, и значение `int`. Функция должна присвоить каждому элементу диапазона массива это значение `int`.
5. Напишите функцию, принимающую имя массива `double` и его размер в качестве аргументов и возвращающую наибольшее значение, которое содержится в этом массиве. Обратите внимание, что функция не должна модифицировать содержимое массива.
6. Почему вы не используете квалификатор `const` для аргументов функций, относящихся к любому из базовых типов?
7. Каковы три формы строк в стиле C могут встретиться в программах C++?

8. Напишите функцию, имеющую следующий прототип:

```
int replace(char * str, char c1, char c2);
```

Эта функция должна заменять каждое появление `c1` в строке `str` на `c2` и возвращать количество выполненных замен.

9. Что означает выражение `*"pizza"`? А как насчет `"taco"[2]`?

10. C++ позволяет передавать структуры по значению, а также передавать адрес структуры. Если `glitz` — структурная переменная, как передать ее по значению? Как передать ее адрес? Каковы преимущества и недостатки обоих подходов?

11. Функция `judge()` имеет тип возврата `int`. В качестве аргумента она принимает адрес функции. Функция, адрес которой ей передается, в свою очередь, принимает аргумент типа `const char` и возвращает `int`. Напишите прототип функции.

12. Предположим, что есть следующее объявление структуры:

```
struct applicant {
    char name[30];
    int credit_ratings[3];
};
```

а. Напишите функцию, которая принимает структуру `applicant` в качестве аргумента и отображает ее содержимое.

б. Напишите функцию, которая принимает адрес структуры `applicant` в качестве аргумента и отображает содержимое структуры, на которую он указывает.

13. Предположим, что функции `f1()` и `f2()` имеют следующие прототипы:

```
void f1(applicant * a);
const char * f2(const applicant * a1, const applicant * a2);
```

Объявите `p1` как указатель на функцию `f1`, а `p2` — как указатель на `f2`. Объявите `ap` как массив из пяти указателей того же типа, что и `p1`, и объявите `pa` как указатель на массив из десяти указателей того же типа, что и `p2`. Воспользуйтесь `typedef`.

Упражнения по программированию

1. Напишите программу, которая многократно запрашивает у пользователя пару чисел до тех пор, пока хотя бы одно из этой пары не будет равно `0`. С каждой парой программа должна использовать функцию для вычисления среднего гармонического этих чисел. Функция должна возвращать ответ `main()` для отображения результата. Среднее гармоническое чисел — это инверсия среднего значения их инверсий; она вычисляется следующим образом:

```
среднее гармоническое = 2.0 * x * y / (x + y)
```

2. Напишите программу, которая запрашивает у пользователя 10 результатов игры в гольф, сохраняя их в массиве. При этом необходимо обеспечить возможность прекращения ввода до ввода всех 10 результатов. Программа должна отобразить все результаты в одной строке и сообщить их среднее значение. Реализуйте ввод, отображение и вычисление среднего в трех отдельных функциях, работающих с массивами.

3. Пусть имеется следующее объявление структуры:

```
struct box
{
    char maker[40];
    float height;
    float width;
    float length;
    float volume;
};
```

   а. Напишите функцию, принимающую структуру `box` по значению и отображающую все ее члены.

   б. Напишите функцию, принимающую адрес структуры `box` и устанавливающую значение члена `volume` равным произведению остальных трех членов.

   в. Напишите простую программу, которая использует эти две функции.

4. Многие лотереи в США организованы подобно той, что была смоделирована в листинге 7.4. Во всех их вариациях вы должны выбрать несколько чисел из одного набора, называемого полем номеров. (Например, вы можете выбрать 5 чисел из поля 1–47.) Вы также указываете один номер (называемый меганомером) из второго диапазона, такого как 1–27. Чтобы выиграть главный приз, вы должны правильно угадать все номера. Шанс выиграть вычисляется как вероятность угадывания всех номеров в поле, умноженная на вероятность угадывания меганомера. Например, вероятность выигрыша в описанном здесь примере вычисляется как вероятность угадывания 5 номеров из 47, умноженная на вероятность угадывания одного номера из 27. Модифицируйте листинг 7.4 для вычисления вероятности выигрыша в такой лотерее.

5. Определите рекурсивную функцию, принимающую целый аргумент и возвращающую его факториал. Вспомните, что факториал 3 записывается, как 3! и вычисляется как 3×2! и т.д., причем 0! равно 1. В общем случае, если `n` больше нуля, то `n! = n * (n-1)!`. Протестируйте функцию в программе, использующей цикл, где пользователь может вводить различные значения, для которых программа вычисляет и отображает факториалы.

6. Напишите программу, использующую описанные ниже функции.

   `Fill_array()` принимает в качестве аргумента имя массива элементов типа `double` и размер этого массива. Она приглашает пользователя ввести значения `double` для помещения их в массив. Ввод прекращается при наполнении массива либо когда пользователь вводит нечисловое значение и возвращает действительное количество элементов.

   `Show_array()` принимает в качестве аргументов имя массива значений `double`, а также его размер, и отображает содержимое массива.

   `Reverse_array()` принимает в качестве аргумента имя массива значений `double`, а также его размер, и изменяет порядок его элементов на противоположный.

   Программа должна использовать эти функции для наполнения массива, обращения порядка его элементов, кроме первого и последнего, с последующим отображением.

7. Вернитесь к программе из листинга 7.7 и замените три функции обработки массивов версиями, которые работают с диапазонами значений, заданными парой указателей. Функция `fill_array()` вместо возврата действительного количества прочитанных значений должна возвращать указатель на место, следующее за последним введенным элементом; прочие функции должны использовать его в качестве второго аргумента для идентификации конца диапазона данных.

8. Вернитесь к программе из листинга 7.15, не использующей класс `array`. Напишите следующие две версии.

   а. Используйте обычный массив из `const char *` для строковых представлений времен года и обычный массив из `double` для расходов.

   б. Используйте обычный массив из `const char *` для строковых представлений времен года и структуру, единственный член которой является обычным массивом из `double` для расходов. (Это очень похоже на базовое проектное решение для класса `array`.)

9. Следующее упражнение позволит попрактиковаться в написании функций, работающих с массивами и структурами. Ниже представлен каркас программы. Дополните его функциями, описанными в комментариях.

```
#include <iostream>
using namespace std;

const int SLEN = 30;
struct student {
    char fullname[SLEN];
    char hobby[SLEN];
    int ooplevel;
};
// getinfo() принимает два аргумента: указатель на первый элемент
// массива структур student и значение int, представляющее
// количество элементов в массиве. Функция запрашивает и
// сохраняет данные о студентах. Ввод прекращается либо после
// наполнения массива, либо при вводе пустой строки в качестве
// имени студента. Функция возвращает действительное количество
// введенных элементов.
int getinfo(student pa[], int n);

// display1() принимает в качестве аргумента структуру student
// и отображает ее содержимое.
void display1(student st);

// display2() принимает адрес структуры student в качестве аргумента
// и отображает ее содержимое.
void display2(const student * ps);

// display3() принимает указатель на первый элемента массива
// структур student и количество элементов в этом массиве и
// отображает содержимое всех структур в массиве.
void display3(const student pa[], int n);

int main()
{
    cout << "Enter class size: ";
    int class_size;
    cin >> class_size;
    while (cin.get() != '\n')
        continue;
```

```
    student * ptr_stu = new student[class_size];
    int entered = getinfo(ptr_stu, class_size);
    for (int i = 0; i < entered; i++)
    {
        display1(ptr_stu[i]);
        display2(&ptr_stu[i]);
    }
    display3(ptr_stu, entered);
    delete [] ptr_stu;
    cout << "Done\n";
    return 0;
}
```

10. Спроектируйте функцию `calculate()`, которая принимает два значения типа `double` и указатель на функцию, принимающую два аргумента `double` и возвращающую значение `double`. Функция `calculate()` также должна иметь тип `double` и возвращать значение, вычисленное функцией, которая задана указателем, используя аргумент `double` функции `calculate()`. Например, предположим, что имеется следующее определение функции `add()`:

```
double add(double x, double y)
{
    return x + y;
}
```

Приведенный ниже вызов функции должен заставить `calculate()` передать значения `2.5` и `10.4` функции `add()` и вернуть ее результат (`12.9`):

```
double q = calculate(2.5, 10.4, add);
```

Используйте в программе эти функции и еще хотя бы одну дополнительную, которая подобна `add()`. В программе должен быть организован цикл, позволяющий пользователю вводить пары чисел. Для каждой пары `calculate()` должна вызвать `add()` и хотя бы еще одну функцию такого рода. Если вы чувствуете себя уверенно, попробуйте создать массив указателей на функции, подобные `add()`, и организуйте цикл, применяя `calculate()` для вызова этих функций по их указателям. Подсказка: вот как можно объявить массив из трех таких указателей:

```
double (*pf[3])(double, double);
```

Инициализировать такой массив можно с помощью обычного синтаксиса инициализации массивов и имен функций в качестве адресов.

8

Дополнительные сведения о функциях

В ЭТОЙ ГЛАВЕ...

- Встроенные функции
- Ссылочные переменные
- Передача функции аргументов по ссылке
- Аргументы по умолчанию
- Перегрузка функций
- Шаблоны функций
- Спецификации шаблонов функций

В главе 7 был представлен обширный материал по функциям, однако осталось еще рассмотреть немало вопросов. Язык C++ предлагает много новых возможностей, связанных с функциями, что отличает его от предшественника — языка C. К ним относятся встроенные функции, передача переменных по ссылке, определяемые по умолчанию значения аргументов, перегрузка функций (полиморфизм) и шаблоны функций. Эта глава более всех предыдущих посвящена специфике языка C++ (но не C), поэтому она служит отправной точкой нашего вторжения в мир "плюсов".

Встроенные функции C++

Встроенные функции являются усовершенствованием языка C++, предназначенным для ускорения работы программ. Основное различие между встроенными и обычными функциями связано не с написанием кода, а с тем, каким образом компилятор внедряет функцию в программу. Чтобы понять это различие, потребуется глубже рассмотреть внутреннее содержание программ. Именно с этого мы и начнем.

Конечным продуктом процесса компиляции является исполняемая программа, которая состоит из набора машинных команд. При запуске программы операционная система загружает эти команды в оперативную память так, что каждая команда обладает собственным адресом в памяти. Затем команды поочередно выполняются. Когда в программе встречается, например, оператор цикла или условного перехода, выполнение "перепрыгивает" вперед или назад через несколько команд, осуществляя переход по определенному адресу. При вызове обычной функции также осуществляется переход к определенному адресу (адресу функции) с последующим возвратом после завершения ее работы. Рассмотрим типичную реализацию этого процесса немного подробнее. Когда в программе встречается команда вызова функции, программа сохраняет адрес команды, следующей сразу после вызова функции, копирует аргументы функции в стек (зарезервированный для этой цели блок памяти), переходит к ячейке памяти, обозначающей начало функции, выполняет код функции (возможно, помещая возвращаемое значение в регистр), а затем переходит к команде, адрес которой сохранен[1]. Переходы и запоминание соответствующих адресов влекут за собой дополнительные затраты времени, связанные с использованием функций.

Встроенные функции C++ предоставляют альтернативу. Скомпилированный код такой функции непосредственно встраивается в код программы. Иначе говоря, компилятор подставляет вместо вызова функции ее код. В результате программе не нужно выполнять переход к другому адресу и возвращаться назад. Таким образом, встраиваемые функции выполняются немного быстрее, чем обычные, однако за это нужно платить дополнительным расходом памяти. Если в десяти различных местах программа выполняет вызов одной и той же встроенной функции, ее код будет содержать десять копий этой функции (рис. 8.1).

Решение об использовании встроенной функции должно быть взвешенным. Если затраты времени на выполнение функции значительно превышают длительность реализации механизма ее вызова, экономия времени на фоне общего процесса окажется незаметной. Если же время выполнения кода невелико, то разница во времени при использовании встроенной функции по сравнению с обычной может оказаться значительной. С другой стороны, в этом случае ускоряется и без того сравнительно быстрый процесс, поэтому при нечастом вызове функции общая экономия времени может быть невелика.

[1] Это можно сравнить с процессом чтения некоторого текста, когда приходится отвлекаться на ознакомление с содержанием сноски, а затем возвращаться к фрагменту, где чтение было прервано.

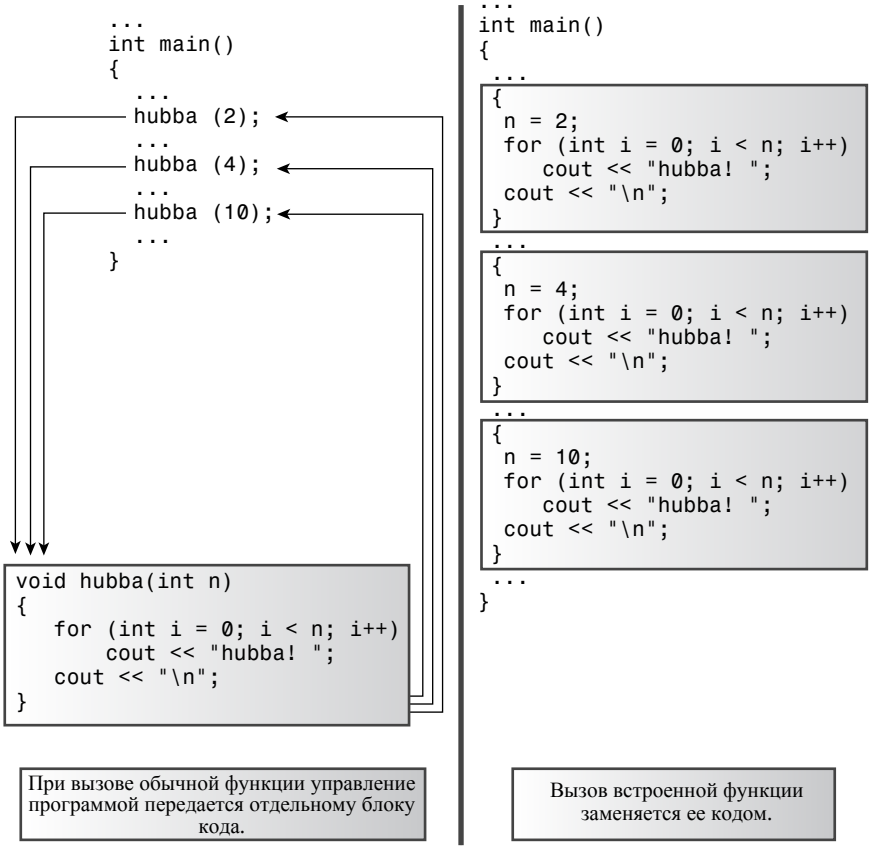


Рис. 8.1. *Различия между встроенными и обычными функциями*

Чтобы воспользоваться встроенной функцией, нужно выполнить хотя бы одно из следующих действий.

- Предварить объявление функции ключевым словом `inline`.
- Предварить определение функции ключевым словом `inline`.

Общепринято опускать прототип и помещать полное описание (заголовок и весь код функции) туда, где обычно находится прототип.

Компилятор не обязательно удовлетворит запрос пользователя на то, чтобы сделать функцию встроенной. Он может прийти к заключению, что функция слишком велика, или обнаружит, что она обращается сама к себе (рекурсия для встроенных функций не допускается и невозможна сама по себе). Кроме того, возможен случай, когда опция реализации встроенных функций у компилятора отключена либо он не поддерживает эту опцию вообще.

В листинге 8.1 иллюстрируется метод встраивания на примере функции `square()`, которая возводит в квадрат переданный ей аргумент. Обратите внимание, что все определение функции уместилось в одну строку. Хотя это не обязательно, но если определение не помещается в одной или двух строках (предполагается, что длинные идентификаторы не используются), то такая функция, скорее всего, является плохим кандидатом на то, чтобы быть встроенной.

Листинг 8.1. inline.cpp

```
// inline.cpp -- использование встроенной функции
#include <iostream>

// Определение встроенной функции
inline double square(double x) { return x * x; }

int main()
{
   using namespace std;
   double a, b;
   double c = 13.0;
   a = square(5.0);
   b = square(4.5 + 7.5);  // допускается передача выражений
   cout << "a = " << a << ", b = " << b << "\n";
   cout << "c = " << c;
   cout << ", c squared = " << square(c++) << "\n";
   cout << "Now c = " << c << "\n";
   return 0;
}
```

Ниже показан вывод программы из листинга 8.1:

```
a = 25, b = 144
c = 13, c squared = 169
Now c = 14
```

Полученные результаты показывают, что встраиваемая функция передает аргументы по значению, как это принято для обычных функций. Если аргумент представляет собой выражение вроде `4.5 + 7.5`, функция передает значение этого выражения. В рассматриваемом случае оно равно `12`. Из этого следует, что средство `inline` языка C++ обладает существенными преимуществами перед макроопределениями языка C. (См. врезку "Встраивание или макросы" ниже в этой главе.)

Хоть в программе нет отдельного прототипа, тем не менее, возможности применения прототипов C++ проявляются и в ней. Дело в том, что полное определение функции, которое дается перед тем, как она будет выполнена первый раз, служит прототипом. Это означает, что можно использовать функцию `square()` с аргументом типа `int` или `long`, и программа автоматически выполнит приведение аргумента к типу `double`, прежде чем передавать его значение функции.

Встраивание или макросы

Средство `inline` появилось только в C++. В языке C используется оператор препроцессора `#define`, обеспечивающий реализацию *макросов*, которые представляют собой грубый аналог встраиваемого кода. Например, макрос, возводящий целое число в квадрат, имеет вид:

```
#define SQUARE(X) X*X
```

Этот макрос работает не по принципу передачи аргументов, а по принципу подстановки текста, при этом `X` играет роль символической метки "аргумента":

`a = SQUARE(5.0);` заменяется на `a = 5.0*5.0;`
`b = SQUARE(4.5 + 7.5);` заменяется на `b = 4.5 + 7.5 * 4.5 + 7.5;`
`d = SQUARE(c++);` заменяется на `d = c++*c++;`

Макрос здесь нормально работает только в первом примере. Положение можно несколько улучшить, снабдив описание макроса скобками:

```
#define SQUARE(X) ((X)*(X))
```

Но все еще остается проблема, связанная с тем, что в макросах не передаются аргументы по значению. Даже при использовании нового определения макроса (со скобками), функция SQUARE(c++) инкрементирует c дважды, в то время как встраиваемая функция square(), представленная в листинге 8.1, вычисляет c, передает полученное значение для возведения в квадрат, а затем инкрементирует c один раз.

Здесь не преследуется цель научить вас создавать макросы на языке C. Вместо этого мы рекомендуем вместо использования макросов для реализации средств, подобных функциям, принять во внимание возможность преобразования их во встроенные функции языка C++.

Ссылочные переменные

Язык C++ вводит в практику новый составной тип данных — ссылочную переменную. *Ссылка* представляет собой имя, которое является псевдонимом, или альтернативным именем, для ранее объявленной переменной. Например, если вы делаете twain ссылкой на переменную clemens, можно взаимозаменяемо использовать эти имена для представления данной переменной. В чем смысл применения альтернативного имени? Не в том ли, чтобы помочь тем программистам, которые не удовлетворены сделанным ими выбором имен переменных? Вполне возможно, однако, основное назначение ссылок — их использование в качестве формальных аргументов функций. Применяя ссылку в качестве аргумента, функция работает с исходными данными, а не с их копиями. Ссылки представляют собой удобную альтернативу указателям при обработке крупных структур посредством функций. Они играют важную роль при создании классов. Однако прежде чем изучать использование ссылок при работе с функциями, рассмотрим основы определения и применения ссылок. Следует иметь в виду, что цель предстоящего обсуждения заключается в демонстрации функционирования ссылок, а не типичных методов их использования.

Создание ссылочных переменных

Как уже упоминалось, в языках C и C++ символ & используется для обозначения адреса переменной. Язык C++ придает символу & дополнительный смысл и задействует его для объявления ссылок. Например, чтобы rodents стало альтернативным именем для переменной rats, необходимо написать следующее:

```
int rats;
int & rodents = rats;   // rodents становится псевдонимом имени rats
```

В таком контексте символ & не является операцией взятия адреса. В этом случае & воспринимается как часть идентификатора типа данных. Подобно тому, как выражение char * в объявлении означает указатель на char, выражение int & представляет собой ссылку на int. Объявление ссылки позволяет взаимозаменяемо использовать идентификаторы rats и rodents. Они ссылаются на одно и то же значение, а также на один и тот же адрес памяти. Программа, представленная в листинге 8.2, подтверждает сказанное.

Листинг 8.2. firstref.cpp

```
// firstref.cpp -- определение и использование ссылки
#include <iostream>
int main()
{
   using namespace std;
   int rats = 101;
   int & rodents = rats; // rodents является ссылкой
```

```
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;
    rodents++;
    cout << "rats = " << rats;
    cout << ", rodents = " << rodents << endl;

    // Некоторые реализации требуют для следующих адресов
    // выполнить приведение к типу unsigned
    cout << "rats address = " << &rats;
    cout << ", rodents address = " << &rodents << endl;
    return 0;
}
```

Обратите внимание, что символ `&` в следующем операторе *не* является операцией взятия адреса, а объявляет, что переменная `rodents` имеет тип `int &` (т.е. является ссылкой на переменную типа `int`):

```
int & rodents = rats;
```

Однако в приведенном ниже операторе символ `&` является операцией взятия адреса, т.е. `&rodents` представляет собой адрес переменной, на которую ссылается `rodents`:

```
cout << ", rodents address = " << &rodents << endl;
```

Вывод программы из листинга 8.2 имеет следующий вид:

```
rats = 101, rodents = 101
rats = 102, rodents = 102
rats address = 0x0065fd48, rodents address = 0x0065fd48
```

Нетрудно заметить, что переменные `rats` и `rodents` имеют одно и то же значение и один и тот же адрес. (Конкретное значение адреса и формат его вывода варьируются от системы к системе.) Инкрементирование `rodents` затрагивает обе перемечные. Точнее, в результате выполнения операции `rodents++` увеличивается на 1 значение единственной переменной, у которой имеется два имени. (Имейте в виду, что хотя этот пример показывает, как действует ссылка, он не может служить образцом типичного ее использования. Обычно ссылка применяется в качестве параметра функции, представляющего, в частности, структуру или объект. Мы вскоре рассмотрим эти виды применения ссылок более подробно.)

На первых порах освоение ссылок программистами, которые работали в C и перешли на C++, не проходит гладко, поскольку ссылки очень напоминают указатели, хотя между ними существуют отличия. Например, можно создать как ссылку, так и указатель, чтобы ссылаться на переменную `rats`:

```
int rats = 101;
int & rodents = rats;      // rodents - ссылка
int * prats = &rats;       // prats - указатель
```

Затем выражения `rodents` и `*prats` могут заменять имя `rats`, а выражения `&rodents` и `prats` могут подменять обозначение `&rats`. С этой точки зрения ссылка во многом подобна указателю в замаскированной нотации, при которой наличие операции разыменования `*` предполагается неявно. И, фактически, это в какой-то степени именно то, чем является ссылка. Однако между ссылками и указателями существуют различия помимо нотации. Одно из таких различий состоит в том, что ссылку необходимо инициализировать в момент ее объявления. Нельзя сначала объявить ссылку, а затем присвоить ей значение, как это делается для указателей:

```
int rat;
int & rodent;
rodent = rat;   // подобное не допускается
```

На заметку!

При объявлении ссылочную переменную необходимо инициализировать.

Ссылка скорее похожа на указатель `const`; ее следует инициализировать в момент создания, и она остается привязанной к определенной переменной до конца программы. Таким образом, конструкция

```
int & rodents = rats;
```

по сути, является замаскированной записью выражения, подобного следующему:

```
int * const pr = &rats;
```

В данном случае ссылка `rodents` играет ту же роль, что и выражение `*pr`.

В листинге 8.3 показано, что произойдет при попытке изменить привязку ссылки с переменной `bunnies` на переменную `rats`.

Листинг 8.3. `secref.cpp`

```
// secref.cpp -- определение и использование ссылки
#include <iostream>
int main()
{
   using namespace std;
   int rats = 101;
   int & rodents = rats;                              // rodents — это ссылка
   cout << "rats = " << rats;
   cout << ", rodents = " << rodents << endl;
   cout << "rats address = " << &rats;
   cout << ", rodents address = " << &rodents << endl; // вывод адресов rats и rodents
   int bunnies = 50;
   rodents = bunnies;                                 // можно ли изменить ссылку?
   cout << "bunnies = " << bunnies;
   cout << ", rats = " << rats;
   cout << ", rodents = " << rodents << endl;
   cout << "bunnies address = " << &bunnies;
   cout << ", rodents address = " << &rodents << endl; // вывод адресов bunnies и rodents
   return 0;
}
```

Ниже показан вывод программы из листинга 8.3:

```
rats = 101, rodents = 101
rats address = 0x0065fd44, rodents address = 0x0065fd44
bunnies = 50, rats = 50, rodents = 50
bunnies address = 0x0065fd48, rodents address = 0x0065fd4
```

Сначала переменная `rodents` ссылается на `rats`, но затем программа предпринимает попытку сделать `rodents` ссылкой на переменную `bunnies`:

```
rodents = bunnies;
```

В какой-то момент кажется, что эта попытка была удачной, поскольку переменная `rodents` вместо значения `101` принимает значение `50`. Однако при ближайшем рассмотрении выясняется, что значение переменной `rats` также изменилось и стало равным `50`. При этом переменные `rats` и `rodents` по-прежнему имеют один и тот же адрес, который отличается от адреса переменной `bunnies`.

Поскольку rodents является псевдонимом переменной rats, оператор присваивания в действительности эквивалентен такому оператору:

```
rats = bunnies;
```

Этот оператор означает следующее: "присвоить переменной rats значение переменной **bunnies**". Короче говоря, ссылку можно устанавливать с помощью инициализирующего объявления, но не операцией присваивания.

Для примера рассмотрим следующий фрагмент кода:

```
int rats = 101;
int * pi = &rats;
int & rodents = *pt;
int bunnies = 50;
pt = &bunnies;
```

Инициализация rodents в *pt приводит к тому, что rodents ссылается на rats. Последующая попытка изменения pt с целью указания на bunnies не отменяет того факта, что rodents ссылается на rats.

Ссылки как параметры функций

Чаще всего ссылки используются в качестве параметров функции, при этом имя переменной в функции становится псевдонимом переменной в вызывающей программе. Такой метод передачи аргументов называется *передачей по ссылке*. Передача параметров по ссылке позволяет вызываемой функции получить доступ к переменным в вызывающей функции. Реализация этого средства в C++ представляет собой дальнейшее развитие основных принципов языка C, где возможна только передача по значению. Вспомните, что передача по значению приводит к тому, что вызываемая функция оперирует копиями значений из вызывающей программы (рис. 8.2).

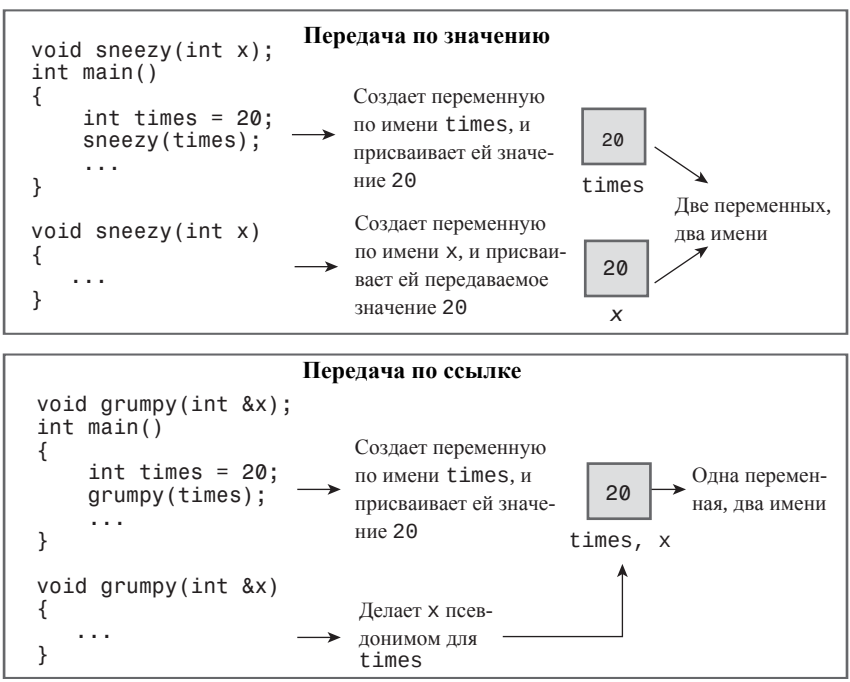


Рис. 8.2. Передача по значению и передача по ссылке

Разумеется, язык C позволяет обойти ограничения, накладываемые передачей аргументов по значению, за счет применения указателей.

Давайте сравним, как используются ссылки и указатели при решении простой задачи: обмен значениями двух переменных. Функция обмена должна иметь возможность изменять значения переменных в вызывающей программе. Это означает, что обычный подход, связанный с передачей переменных по значению, здесь не подойдет, поскольку функция выполнит обмен содержимым лишь копий исходных переменных, но не их самих. Однако если передавать ссылки, функция получит возможность работать с исходными данными. Вместо этого для получения доступа к исходным данным можно передавать указатели. В листинге 8.4 демонстрируются все три метода, включая и тот, который не дает желаемого результата, так что вы можете легко сравнить их.

Листинг 8.4. swaps.cpp

```
// swaps.cpp — обмен значениями с помощью ссылок и указателей
#include <iostream>
void swapr(int & a, int & b);           // a, b - псевдонимы для int
void swapp(int * p, int * q);           // p, q — адреса int
void swapv(int a, int b);               // a, b - новые переменные
int main()
{
   using namespace std;
   int wallet1 = 300;
   int wallet2 = 350;
   cout << "wallet1 = $" << wallet1;
   cout << " wallet2 = $" << wallet2 << endl;

   // Использование ссылок для обмена содержимого
   cout << "Using references to swap contents:\n";
   swapr(wallet1, wallet2);             // передача переменных
   cout << "wallet1 = $" << wallet1;
   cout << " wallet2 = $" << wallet2 << endl;

   // Использование указателей для обмена содержимого
   cout << "Using pointers to swap contents again:\n";
   swapp(&wallet1, &wallet2);           // передача адресов переменных
   cout << "wallet1 = $" << wallet1;
   cout << " wallet2 = $" << wallet2 << endl;

   // Попытка использования передачи по значению
   cout << "Trying to use passing by value:\n";
   swapv(wallet1, wallet2);             // передача значений переменных
   cout << "wallet1 = $" << wallet1;
   cout << " wallet2 = $" << wallet2 << endl;
   return 0;
}
void swapr(int & a, int & b)            // использование ссылок
{
   int temp;
   temp = a;              // использование a, b для получения значений переменных
   a = b;
   b = temp;
}
void swapp(int * p, int * q)            // использование указателей
{
   int temp;
   temp = *p;             // использование *p, *q для получения значений переменных
   *p = *q;
   *q = temp;
}
```

```
void swapv(int a, int b)  // попытка использования значений
{
   int temp;
   temp = a;              // использование a, b для получения значений переменных
   a = b;
   b = temp;
}
```

Ниже показан вывод программы из листинга 8.4:

```
wallet1 = $300 wallet2 = $350          ← исходные значения
Using references to swap contents:
wallet1 = $350 wallet2 = $300          ← обмен значениями выполнен
Using pointers to swap contents again:
wallet1 = $300 wallet2 = $350          ← обмен значениями снова выполнен
Trying to use passing by value:
wallet1 = $300 wallet2 = $350          ← обмен значениями не удался
```

Как и ожидалось, методы, использующие указатели и ссылки, успешно реализовали обмен содержимым, в то время как метод передачи по значению завершился неудачей.

Замечания по программе

Прежде всего, обратите внимание на то, как вызывается каждая функция в листинге 8.4:

```
swapr(wallet1, wallet2);      // передача переменных
swapp(&wallet1, &wallet2);    // передача адресов переменных
swapv(wallet1, wallet2);      // передача значений переменных
```

Передача по ссылке (`swapr(wallet1, wallet2)`) и передача по значению (`swapv(wallet1, wallet2)`) выглядят идентично. Единственный способ определить, что функция `swapr()` передает аргументы по ссылке — обратиться к прототипу или определению функции. В то же время, наличие операции взятия адреса (`&`) явно говорит о том, что функции передается адрес значения (`swapp(&wallet1, &wallet2)`). (Вспомните, что объявление типа `int *p` означает, что `p` — это указатель на `int`, поэтому аргумент, соответствующий `p`, должен быть адресом, таким как `&wallet1`.)

Далее сравним программный код функций `swapr()` (передача по ссылке) и `swapv()` (передача по значению). Единственное видимое различие между ними связано с объявлением параметров:

```
void swapr(int & a, int & b)
void swapv(int a, int b)
```

Внутреннее различие между ними, естественно, состоит в том, что в функции `swapr()` переменные `a` и `b` служат псевдонимами имен `wallet1` и `wallet2`, так что обмен значениями между `a` и `b` вызывает обмен значениями между переменными `wallet1` и `wallet2`. В то же время в функции `swapv()` переменные `a` и `b` — это новые перемснные, которые копируют значения переменных `wallet1` и `wallet2`. В этом случае обмен значениями между `a` и `b` никак не влияет на переменные `wallet1` и `wallet2`.

И, наконец, сравним функцию `swapr()` (передача ссылки) и `swapp()` (передача указателя). Первое различие кроется в объявлении параметров:

```
void swapr(int & a, int & b)
void swapp(int * p, int * q)
```

Второе различие состоит в том, что вариант с указателем требует применения операции разыменования (`*`) во всех случаях, когда функция использует переменные `p` и `q`.

Как уже упоминалось ранее, ссылочную переменную необходимо инициализировать при ее определении. Вызов функции инициализирует свои параметры значениями аргументов, передаваемых в вызове. Это значит, что следующий вызов функции инициализирует формальный параметр `a` значением `wallet1`, а формальный параметр `b` — значением `wallet2`:

```
swapr(wallet1, wallet2);
```

Свойства и особенности ссылок

С использованием ссылочных аргументов связан ряд особенностей, о которых следует знать. Для начала обратимся к листингу 8.5. В нем используются две функции для возведения в куб значения аргумента. Одна из них принимает аргумент типа `double`, в то время как другая получает ссылку на значение типа `double`. Код возведения в куб преднамеренно выглядит несколько необычно, чтобы проиллюстрировать работу со ссылками.

Листинг 8.5. `cubes.cpp`

```
// cubes.cpp -- обычные и ссылочные аргументы
#include <iostream>
double cube(double a);
double refcube(double &ra);
int main ()
{
   using namespace std;
   double x = 3.0;
   cout << cube(x);
   cout << " = cube of " << x << endl;         // вывод значения в кубе
   cout << refcube(x);
   cout << " = cube of " << x << endl;         // вывод значения в кубе
   return 0;
}
double cube(double a)
{
   a *= a * a;
   return a;
}
double refcube(double &ra)
{
   ra *= ra * ra;
   return ra;
}
```

Ниже показан вывод программы из листинга 8.5:

```
27 = cube of 3
27 = cube of 27
```

Обратите внимание, что функция `refcube()` изменяет значение `x` в функции `main()`, в то время как функция `cube()` этого не делает. Это напоминает причину, почему передача по значению является нормой. Переменная `a` является локальной для функции `cube()`. Она инициализируется значением `x`, однако изменения переменной `a` не отражаются на `x`. Тем не менее, поскольку функция `refcube()` использует в качестве аргумента ссылку, изменения, которые она вносит в переменную `ra`, фактически выполняются над переменной `x`. Если требуется, чтобы функция использовала переда-

ваемую ей информацию, но не изменяла ее, и при этом использовать ссылку, следует воспользоваться постоянной ссылкой. В рассматриваемом примере в прототипе и заголовке функции понадобилось бы применить квалификатор `const`:

```
double refcube(const double &ra);
```

Но в таком случае компилятор выводил бы сообщение об ошибке всякий раз, когда обнаруживал бы код, изменяющий значение переменной `ra`.

Если потребуется создать функцию с использованием идеи из рассматриваемого примера (т.е. применение базового числового типа), нужно выполнить передачу аргумента по значению, а не более экзотичную передачу по ссылке. Ссылочные аргументы полезно применять для более крупных элементов данных, таких как структуры и классы, в чем вы вскоре убедитесь.

Функции, которые осуществляют передачу данных по значению, такие как `cube()` из листинга 8.5, могут использовать множество видов аргументов. Например, все приведенные ниже вызовы допустимы:

```
double z = cube(x + 2.0);       // вычисление выражения x + 2.0,
                                // передача значения
z = cube(8.0);                  // передача значения 8.0
int k = 10;
z = cube(k);                    // преобразование значения k в double,
                                // передача значения
double yo[3] = { 2.2, 3.3, 4.4 };
z = cube (yo[2]);               // передача значения 4.4
```

Предположим, что вы пробуете использовать аналогичные аргументы для функции со ссылочными параметрами. Создается впечатление, что передача ссылки должна быть более ограниченной. В конце концов, если `ra` является альтернативным именем переменной `a`, то фактическим аргументом должна быть именно эта переменная. Показанный ниже оператор не выглядит имеющим смысл, поскольку выражение `x + 3.0` не является переменной:

```
double z = refcube(x + 3.0);    // может привести к ошибке компиляции
```

Например, нельзя присвоить значение следующему выражению:

```
x + 3.0 = 5.0;                  // не имеет смысла
```

Что произойдет при попытке выполнить обращение к функции наподобие такого: `refcube(x + 3.0)`? В современном языке C++ это ошибка, и большинство компиляторов выведут сообщение об этом. Другие отобразят предупреждение примерно такого содержания:

```
Warning: Temporary used for parameter 'ra' in call to refcube(double &)
```

Предупреждение: при вызове refcube(double &) для параметра 'ra' используется временная переменная

Причина такой не слишком категоричной формулировки в том, что язык C++ в годы своего становления допускал передачу выражений в качестве ссылочных переменных. В некоторых случаях это разрешено и сейчас. А происходит вот что: поскольку `x + 3.0` не является переменной типа `double`, программа создает временную переменную, не имеющую имени, и инициализирует ее значением выражения `x + 3.0`. Затем `ra` становится ссылкой на эту временную переменную. Давайте рассмотрим временные переменные более подробно и выясним, в каких случаях они создаются и в каких — нет.

Временные переменные, ссылочные аргументы и квалификатор const

C++ может генерировать временную переменную, если фактический аргумент не соответствует ссылочному аргументу. В настоящее время C++ допускает это только в случае, когда аргументом является ссылка с квалификатором const, но это не всегда так. Рассмотрим случаи, когда C++ генерирует временную переменную, и выясним, почему ограничение, требующее ссылки const, имеет смысл.

Прежде всего, в каких случаях создается временная переменная? При условии, что ссылочный параметр является const, компилятор генерирует временную переменную в двух ситуациях:

- когда тип фактического аргумента выбран правильно, но сам параметр не является lvalue;
- когда тип фактического параметра выбран неправильно, но может быть преобразован в правильный тип.

Что такое lvalue? Аргумент, являющийся lvalue, представляет собой объект данных, на который можно ссылаться по адресу. Например, переменная, элемент массива, член структуры, ссылка и разыменованный указатель — все они являются lvalue. К lvalue не относятся литеральные константы (кроме строк в двойных кавычках, которые представлены своими адресами) и выражения, состоящие из нескольких элементов. Понятие *lvalue* в C первоначально означало сущности, которые могли находиться в левой части оператора присваивания, но это было до появления ключевого слова const. Теперь как обычная, так и переменная const могут рассматриваться как lvalue, поскольку к ним обеим можно обращаться по адресу. Вдобавок обычная переменная может быть дополнительно определена как *изменяемое lvalue*, а переменная const — как *неизменяемое lvalue*.

Вернувшись к рассматриваемому примеру, предположим, что мы переопределили функцию refcube() так, что у нее имеется аргумент в виде ссылочной константы:

```
double refcube(const double &ra)
{
   return ra * ra * ra;
}
```

Теперь взгляните на следующий код:

```
double side = 3.0;
double * pd = &side;
double & rd = side;
long edge = 5L;
double lens[4] = { 2.0, 5.0, 10.0, 12.0 };
double c1 = refcube(side);              // ra - это side
double c2 = refcube(lens[2]);           // ra - это lens[2]
double c3 = refcube(rd);                // ra - это rd, которая side
double c4 = refcube(*pd)                // ra - это *pd, которая side
double c5 = refcube(edge);              // ra - временная переменная
double c6 = refcube(7.0);               // ra - временная переменная
double c7 = refcube(side + 10.0);       // ra - временная переменная
```

Аргументы side, lens[2], rd и *pd являются именованными объектами данных типа double, поэтому для них есть возможность сгенерировать ссылки, так что временные переменные не нужны. (Вспомните, что элемент массива ведет себя подобно переменной с тем же типом, что и элемент.) Но хотя объект edge и является перемен-

ной, тип ее не подходит. Ссылка на объект типа `double` не может ссылаться на данные типа `long`. С другой стороны, аргументы `7.0` и `side + 10.0` имеют правильный тип, но не являются именованными объектами данных. В каждом из этих случаев компилятор генерирует временную анонимную переменную и заставляет `ra` ссылаться на нее. Такие временные переменные существуют во время вызова функции, после чего компилятор может уничтожить их.

Так почему же такое поведение оправдано для константных ссылок и недопустимо в других случаях? Вернемся к функции `swapr()` из листинга 8.4:

```
void swapr(int & a, int & b)  // использование ссылок
{
    int temp;
    temp = a;   // использование a, b для получения значений переменных
    a = b;
    b = temp;
}
```

Что произойдет, если выполнить представленный ниже программный код в условиях менее жестких правил ранних версий C++?

```
long a = 3, b = 5;
swapr(a, b);
```

Здесь имеет место несоответствие типов, поэтому компилятор создает две временных переменных типа `int`, инициализирует их значениями `3` и `5`, а затем производит обмен содержимым временных переменных, оставляя при этом значения `a` и `b` неизменными.

Короче говоря, если назначение функции со ссылочными аргументами состоит в том, чтобы модифицировать переменные, передаваемые в качестве аргументов, ситуация, которую создают временные переменные, препятствуют достижению этой цели. Решение заключается в том, чтобы запретить создание временных переменных в таких ситуациях, и новый стандарт C++ реализует именно этот подход. (Однако некоторые компиляторы по-прежнему выводят предупреждение вместо сообщения об ошибке, поэтому если вы получили предупреждение об использовании временных переменных, ни в коем случае не игнорируйте его.)

Теперь рассмотрим функцию `refcube()`. В ее задачу входит простое использование переданных значений без их модификации. В этом случае временные переменные не приносят никакого вреда; они придают функции более универсальный характер в отношении разнообразия аргументов, с которыми она способна работать. Следовательно, если объявление функции указывает, что ссылка имеет тип `const`, C++ при необходимости генерирует временные переменные. По сути, функция C++, принимающая формальный ссылочный аргумент с квалификатором `const` и с несоответствующим фактическим аргументом, имитирует традиционные действия, выполняемые при передаче аргументов по значению. При этом гарантируется, что исходные данные не подвергнутся изменениям, а для хранения соответствующего значения применяется временная переменная.

На заметку!

Если передаваемый функции аргумент не является lvalue или не совместим по типу с соответствующим ссылочным параметром `const`, C++ создает анонимную переменную требуемого типа, присваивает ей значение передаваемого функции аргумента, и делает так, чтобы параметр ссылался на эту переменную.

Используйте const, когда это возможно

Существуют три серьезных причины объявлять ссылочные аргументы как ссылки на константные данные.

- Использование `const` защищает от внесения в программы ошибок, приводящих к непреднамеренному изменению данных.
- Использование `const` позволяет функции обрабатывать фактические аргументы как с `const`, так и без `const`. При этом функция, в прототипе которой квалификатор `const` опущен, может принимать только неконстантные данные.
- Использование ссылки `const` позволяет функции генерировать и использовать временные переменные по мере необходимости.

Формальные ссылочные аргументы рекомендуется объявлять с квалификатором `const` во всех случаях, когда для этого есть возможность.

В C++11 появилась вторая разновидность ссылки — *ссылка rvalue*, которая может ссылаться на rvalue. Она объявляется с применением `&&`:

```
double && rref = std::sqrt(36.00);      // не разрешено для double &
double j = 15.0;
double && jref = 2.0* j + 18.5;         // не разрешено для double &
std::cout << rref << '\n';              // отображает 6.0
std::cout << jref << '\n';              // отображает 48.5;
```

Ссылка rvalue была введена в основном для того, чтобы помочь разработчикам библиотек предоставлять более эффективные реализации определенных операций. В главе 18 будет показано, как использовать ссылки rvalue для реализации подхода, который называется семантикой переноса. Исходный ссылочный тип (объявленный с использованием `&`) теперь называется ссылкой lvalue.

Использование ссылок при работе со структурами

Ссылки очень хорошо работают со структурами и классами, т.е. с типами данных C++, определяемыми пользователем. Собственно говоря, ссылки были введены, прежде всего, для использования именно с этими типами, а не с базовыми встроенными типами данных.

Метод использования ссылки на структуру в качестве параметра функции ничем не отличается от метода применения ссылки на базовую переменную: при объявлении параметра структуры достаточно воспользоваться операцией ссылки `&`. Например, предположим, что есть следующее определение структуры:

```
struct free_throws
{
   std::string name;
   int made;
   int attempts;
   float percent;
};
```

Затем функция, использующая ссылку на этот тип, может иметь такой прототип:

```
void set_pc(free_throws & ft);          // использование ссылки на структуру
```

Если функция не должна изменять структуру, необходимо применить `const`:

```
void display(const free_throws & ft);  // не разрешать изменения структуры
```

Программа из листинга 8.6 именно это и делает. Кроме того, она реализует интересную идею — функция возвращает ссылку на структуру. Такое поведение несколько отличается от случая, когда функция возвращает структуру. Потребуется принять определенные меры предосторожности, которые вскоре будут рассмотрены.

Листинг 8.6. strc_ref.cpp

```
// strc_ref.cpp -- использование ссылок на структуру
#include <iostream>
#include <string>
struct free_throws
{
    std::string name;
    int made;
    int attempts;
    float percent;
};
void display(const free_throws & ft);
void set_pc(free_throws & ft);
free_throws & accumulate(free_throws & target, const free_throws & source);

int main()
{
    // Частичные инициализации — оставшиеся неинициализированными
    // члены устанавливаются в 0
    free_throws one = {"Ifelsa Branch", 13, 14};
    free_throws two = {"Andor Knott", 10, 16};
    free_throws three = {"Minnie Max", 7, 9};
    free_throws four = {"Whily Looper", 5, 9};
    free_throws five = {"Long Long", 6, 14};
    free_throws team = {"Throwgoods", 0, 0};

    // Инициализация не производится
    free_throws dup;
    set_pc(one);
    display(one);
    accumulate(team, one);
    display(team);

    // Использование возвращаемого значения в качестве аргумента
    display(accumulate(team, two));
    accumulate(accumulate(team, three), four);
    display(team);

    // Использование возвращаемого значения в присваивании
    dup = accumulate(team,five);
    std::cout << "Displaying team:\n";
    display(team);

    // Отображение dup после присваивания
    std::cout << "Displaying dup after assignment:\n";
    display(dup);
    set_pc(four);

    // Неблагоразумное присваивание
    accumulate(dup,five) = four;

    // Отображение dup после неблагоразумного присваивания
    std::cout << "Displaying dup after ill-advised assignment:\n";
    display(dup);
    return 0;
}
```

```
void display(const free_throws & ft)
{
   using std::cout;
   cout << "Name: " << ft.name << '\n';           // вывод члена name
   cout << " Made: " << ft.made << '\t';          // вывод члена made
   cout << "Attempts: " << ft.attempts << '\t';   // вывод члена attempts
   cout << "Percent: " << ft.percent << '\n';     // вывод члена percent
}
void set_pc(free_throws & ft)
{
   if (ft.attempts != 0)
      ft.percent = 100.0f *float(ft.made)/float(ft.attempts);
   else
      ft.percent = 0;
}
free_throws & accumulate(free_throws & target, const free_throws & source)
{
   target.attempts += source.attempts;
   target.made += source.made;
   set_pc(target);
   return target;
}
```

Ниже показан вывод программы из листинга 8.6:

```
Name: Ifelsa Branch
 Made: 13      Attempts: 14    Percent: 92.8571
Name: Throwgoods
 Made: 13      Attempts: 14    Percent: 92.8571
Name: Throwgoods
 Made: 23      Attempts: 30    Percent: 76.6667
Name: Throwgoods
 Made: 35      Attempts: 48    Percent: 72.9167
Displaying team:
Name: Throwgoods
 Made: 41      Attempts: 62    Percent: 66.129
Displaying dup after assignment:
Name: Throwgoods
 Made: 41      Attempts: 62    Percent: 66.129
Displaying dup after ill-advised assignment:
Name: Whily Looper
 Made: 5       Attempts: 9     Percent: 55.5556
```

Замечания по программе

Эта программа начинается с инициализации нескольких структурных объектов. Вспомните, что если инициализаторов меньше, чем членов, остальные члены (как `percent` в данном случае) устанавливаются в 0. Первый вызов функции выглядит следующим образом:

```
set_pc(one);
```

Поскольку формальный параметр `ft` в `set_pc()` является ссылкой, `ft` ссылается на `one`, и код в `set_pc()` устанавливает член `one.percent`. Передача по значению в этом случае работать не будет, поскольку это приведет к установке члена `percent` временной копии `one`. Альтернатива, как говорилось в предшествующей главе, заключается

в использовании параметра типа указателя и передаче адреса, но форма становится более сложной:

```
set_pcp(&one);       // использование указателей - &one вместо one
...
void set_pcp(free_throws * pt)
{
   if (pt->attempts != 0)
       pt->percent = 100.0f *float(pt->made)/float(pt->attempts);
   else
       pt->percent = 0;
}
```

Рассмотрим следующий вызов функции:

```
display(one);
```

Поскольку `display()` отображает содержимое структуры, не изменяя его, в этой функции применяется ссылочный параметр `const`. В таком случае структуру можно было бы передать по значению, однако использование ссылки более экономично с точки зрения времени и памяти, чем создание копии исходной структуры.

Далее следует такой вызов функции:

```
accumulate(team, one);
```

Функция `accumulate()` принимает два структурных аргумента. Она добавляет значения членов `attempts` и `made` второй структуры к соответствующим членам первой структуры. Изменяется только первая структура, поэтому первый параметр является ссылкой, тогда как второй — ссылкой `const`:

```
free_throws & accumulate(free_throws & target, const free_throws & source);
```

А что с возвращаемым значением? В только что рассмотренном вызове функции оно не используется; в таком случае функция могла бы иметь тип `void`. Но посмотрите на следующий вызов этой же функции:

```
display(accumulate(team, two));
```

Что здесь происходит? Давайте обратимся к объекту структуры `team`. Сначала `team` передается функции `accumulate()` в качестве первого аргумента. Это значит, что объект `target` в `accumulate()` в действительности является `team`. Функция `accumulate()` модифицирует `team`, после чего возвращает его в виде ссылки. Обратите внимание, что оператор возврата в функции выглядит так:

```
return target;
```

В этом операторе ничего не указывает на то, что возвращается ссылка. Необходимая информация поступает из заголовка функции (а также из прототипа):

```
free_throws & accumulate(free_throws & target, const free_throws & source)
```

Если бы в качестве возвращаемого типа был объявлен `free_throws`, а не `free_throws &`, тот же самый оператор возврата вернул бы копию `target` (и, следовательно, копию `team`). Но типом возврата является ссылка, поэтому возвращаемым значением будет исходный объект `team`, переданный первым в `accumulate()`.

А что произойдет дальше? Возвращаемое значение `accumulate()` — первый аргумент в вызове `display()`, а это значит, что в качестве первого аргумента в `display()` передается `team`. Поскольку параметром `display()` является ссылка, объектом `ft` в `display()` в действительности будет `team`. Следовательно, отобразится содержимое `team`.

Совокупный эффект от вызова

```
display(accumulate(team, two));
```

будет тем же самым, что и от следующих вызовов:

```
accumulate(team, two);
display(team);
```

Аналогичная логика применима и к показанному ниже оператору:

```
accumulate(accumulate(team, three), four);
```

Он дает тот же эффект, что и следующие операторы:

```
accumulate(team, three);
accumulate(team, four);
```

Далее в программе идет оператор присваивания:

```
dup = accumulate(team,five);
```

Как и можно было предположить, он копирует значения из `team` в `dup`.

Наконец, в программе еще раз используется функция `accumulate()`, но не в характерной для нее манере:

```
accumulate(dup,five) = four;
```

Этот оператор — т.е. присваивание значения вызову функции — работает потому, что возвращаемое значение является ссылкой. Если бы в `accumulate()` применялся возврат по значению, такой код не скомпилировался бы. Поскольку возвращаемое значение — это ссылка на `dup`, этот код имеет тот же самый эффект, что и следующие операторы:

```
accumulate(dup,five);     // добавить данные five к dup
dup = four;               // перезаписать содержимое dup содержимым four
```

Второй оператор затирает работу, выполненную первым оператором, так что приведенный выше оператор присваивания не может служить примером адекватного использования функции `accumulate()`.

Зачем возвращать ссылку?

Давайте посмотрим более внимательно, чем возврат ссылки отличается от традиционного механизма возврата. Работа последнего очень похожа на передачу по значению параметров функции. Выражение, следующее за `return`, вычисляется, и полученное значение передается обратно вызывающей функции. Концептуально это значение копируется во временную ячейку и вызывающая программа его использует. Рассмотрим следующий код:

```
double m = sqrt(16.0);
cout << sqrt(25.0);
```

В первом операторе значение `4.0` копируется во временную ячейку, после чего значение из этой ячейки копируется в `m`. Во втором операторе значение `5.0` копируется во временную ячейку и затем содержимое этой ячейки передается в `cout`. (Это концептуальное описание. На практике оптимизирующий компилятор может объединять некоторые шаги.)

Теперь рассмотрим следующий оператор:

```
dup = accumulate(team,five);
```

Если accumulate() будет возвращать структуру вместо ссылки на структуру, это может повлечь за собой копирование целой структуры во временную ячейку и последующее копирование этой копии в dup. Но благодаря ссылочному возвращаемому значению, team копируется напрямую в dup, что является более эффективным подходом.

На заметку!

Функция, которая возвращает ссылку, фактически является псевдонимом переменной, на которую ссылается.

Будьте осмотрительными при выборе объекта, на который указывает возвращаемая ссылка

Важнее всего избегать ситуации, когда возвращаемая ссылка указывает на область памяти, которая прекращает существование после завершения работы функции. Ниже приведен пример того, как поступать не следует:

```
const free_throws & clone2(free_throws & ft)
{
   free_throws newguy;          // первый шаг к серьезной ошибке
   newguy = ft;                 // копирование информации
   return newguy;               // возврат ссылки на копию
}
```

В результате выполнения этого кода возвращается ссылка на временную переменную (newguy), которая прекращает существование сразу после завершения работы функции. (Время жизни переменных различного вида обсуждается в главе 9.) Подобно этому следует избегать возврата указателей на такие временные переменные.

Проще всего избежать такой ошибки за счет возврата ссылки, которая была передана функции в качестве аргумента. Ссылочный параметр будет ссылаться на данные, используемые вызывающей функцией; таким образом, возвращаемая ссылка будет ссылаться на те же данные. Это, например, делается в функции accumulate() из листинга 8.6.

Второй метод заключается в использовании операции new для создания нового хранилища. Ранее уже рассматривались примеры, в которых с помощью new создавалось пространство для строки, а функция возвращала указатель на это пространство. Вот как решается подобная задача с помощью ссылки:

```
const free_throws & clone(free_throws & ft)
{
   free_throws * pt;
   *pt = ft;                    // копирование информации
   return *pt;                  // возврат ссылки на копию
}
```

Первый оператор создает безымянную структуру free_throws. Указатель pt указывает на эту структуру, таким образом, *pt — это сама структура. Создается впечатление, что приведенный выше код возвращает структуру, однако объявление функции отражает, что она возвращает ссылку на эту структуру. Затем функцию можно использовать следующим образом:

```
free_throws & jolly = clone(three);
```

Это делает jolly ссылкой на новую структуру. С таким подходом связана одна проблема: потребуется использовать оператор delete для освобождения памяти, выделенной операцией new, когда она больше не нужна. Вызов функции call() маскирует обращение к new, поэтому легко забыть о применении delete впоследствии. Шаблон

auto_ptr или более эффективный шаблон unique_prt из C++11, который рассматривается в главе 16, может помочь автоматизировать процесс освобождения памяти.

Причины использования квалификатора const при объявлении возвращаемой ссылки

Вспомните, что в листинге 8.6 присутствовал следующий оператор:

```
accumulate(dup,five) = four;
```

Его эффект состоит в том, что сначала добавляются данные из five в dup, а затем содержимое dup перезаписывается содержимым four. Почему этот оператор скомпилировался? Присваивание требует в левой части изменяемого lvalue. То есть подвыражение слева от знака присваивания должно идентифицировать блок памяти, который можно модифицировать. В этом случае функция возвращает ссылку на dup, которая как раз идентифицирует такой блок памяти. Таким образом, этот оператор вполне допустим.

Обычные (не ссылочные) возвращаемые типы, с другой стороны, являются rvalue, т.е. значениями, к которым нельзя обратиться по адресу. Такие выражения могут находиться в правой части оператора присваивания, но не в левой. Другие примеры rvalue включают литералы, такие как 10.0, и выражения наподобие x + y. Очевидно, что не имеет смысл пытаться получить адрес литерала, такого как 10.0, но почему тогда нормальным возвращаемым значением функции является rvalue? Причина в том, что возвращаемое значение, как вы помните, хранится во временной ячейке памяти, которая не обязательно будет существовать на момент выполнения следующего оператора.

Предположим, что нужно использовать ссылочное возвращаемое значение, но не допускать такого поведения, как присваивание значения функции accumulate(). Для этого просто сделайте возвращаемый тип ссылкой const:

```
const free_throws &
   accumulate(free_throws & target, const free_throws & source);
```

Теперь возвращаемый тип является const, следовательно, неизменяемым lvalue. Таким образом, показанное ниже присваивание больше не разрешается:

```
accumulate(dup,five) = four;  // не разрешено для возврата ссылки const
```

А как с другими вызовами этой функции в программе? Следующий оператор остается допустимым даже со ссылочным возвращаемым типом const:

```
display(accumulate(team, two));
```

Это объясняется тем, что формальный параметр display() также имеет тип const free_thows &. Однако показанный ниже оператор не будет разрешен, поскольку первый формальный параметр accumulate() не является const:

```
accumulate(accumulate(team, three), four);
```

Существенная ли это потеря? Не в этом случае, т.к. по-прежнему можно поступать следующим образом:

```
accumulate(team, three);
accumulate(team, four);
```

И, разумеется, по-прежнему можно использовать accumulate() в правой части оператора присваивания.

Опуская const, можно создавать более краткий, но более сложный для понимания код.

Как правило, следует избегать конструкций, смысл которых неочевиден, потому что они повышают вероятность внесения сложных для выявления ошибок. Таким образом, применение в качестве возвращаемого значения ссылки `const` помогает преодолеть искушение внести путаницу. Однако иногда имеет смысл не указывать `const`. Примером может служить перегруженная операция <<, которая рассматривается в главе 11.

Использование ссылок на объект класса

В языке C++ для передачи функциям объектов классов обычно практикуется использование ссылок. Например, ссылочные параметры применяются в функциях, принимающих объекты классов `string`, `ostream`, `istream`, `ofstream` и `ifstream` в качестве аргументов.

Рассмотрим пример, в котором используется класс `string`, а также демонстрируются различные решения, в том числе неудачные. Замысел состоит в том, чтобы создать функцию, которая добавляет заданную строку к обеим сторонам другой строки. В листинге 8.7 представлены три функции, предназначенные для решения этой задачи. Однако одно из решений настолько ошибочное, что может привести к сбою программы или даже отказу компиляции.

Листинг 8.7. `strquote.cpp`

```
// strquote.cpp -- различные решения
#include <iostream>
#include <string>
using namespace std;
string version1(const string & s1, const string & s2);
const string & version2(string & s1, const string & s2); // имеет побочный
                                                         // эффект
const string & version3(string & s1, const string & s2);  // неудачное решение
int main()
{
    string input;
    string copy;
    string result;
    cout << "Enter a string: ";
    getline(cin, input);              // ввод строки
    copy = input;
    cout << "Your string as entered: " << input << endl;
    result = version1(input, "***"); // отображение выведенной строки
    cout << "Your string enhanced: " << result << endl;
          // вывод расширенной строки
    cout << "Your original string: " << input << endl;
          // вывод исходной строки
    result = version2(input, "###");
    cout << "Your string enhanced: " << result << endl;
          // вывод расширенной строки
    cout << "Your original string: " << input << endl;
          // вывод исходной строки
    cout << "Resetting original string.\n";
          // восстановление исходной строки
    input = copy;
    result = version3(input, "@@@");
    cout << "Your string enhanced: " << result << endl;
          // вывод расширенной строки
    cout << "Your original string: " << input << endl;  // вывод исходной строки
    return 0;
}
```

```
string version1(const string & s1, const string & s2)
{
    string temp;
    temp = s2 + s1 + s2;
    return temp;
}
const string & version2(string & s1, const string & s2) // имеет побочный эффект
{
    s1 = s2 + s1 + s2;
    // Возврат ссылки, переданной функции, безопасен
    return s1;
}
const string & version3(string & s1, const string & s2) // неудачное решение
{
    string temp;
    temp = s2 + s1 + s2;
    // Возврат ссылки на локальную переменную небезопасен
    return temp;
}
```

Ниже показан пример выполнения программы из листинга 8.7:

```
Enter a string: It's not my fault.
Your string as entered: It's not my fault.
Your string enhanced: ***It's not my fault.***
Your original string: It's not my fault.
Your string enhanced: ###It's not my fault.###
Your original string: ###It's not my fault.###
Resetting original string.
```

В этой точке программа дает сбой.

Замечания по программе

Первая версия функции в листинге 8.7 наиболее прямолинейна:

```
string version1(const string & s1, const string & s2)
{
    string temp;

    temp = s2 + s1 + s2;
    return temp;
}
```

Функция принимает два аргумента типа `string` и использует класс `string` для создания новой строки, которая обладает требуемыми свойствами. Обратите внимание, что оба аргумента функции представляют собой ссылки `const`. Результат работы функции такой же, как если бы ей передавались два объекта типа `string`:

```
string version4(string s1, string & s2)  // дает тот же результат
```

В этом случае аргументы `s1` и `s2` — это новые объекты `string`. Таким образом, применять ссылки эффективнее, поскольку в этом случае функция не будет создавать новые объекты и копировать в них данные из исходных объектов. Здесь квалификатор `const` указывает, что функция использует, но не изменяет исходные строки.

Объект `temp` является новым и локальным в функции `version1()`. Он прекращает существование после завершения работы функции. Таким образом, возврат объекта `temp` в качестве ссылки невозможен, поэтому для функции задан тип `string`. Это означает, что содержимое объекта `temp` будет скопировано во временную область хране-

ния возвращаемых значений. Затем в функции `main()` содержимое области хранения копируется в строку по имени `result`:

```
result = version1(input, "***");
```

Передача аргумента — строки в стиле C параметру — ссылке на объект `string`

Вы могли отметить интересную особенность функции `version1()`: для обоих формальных параметров (`s1` и `s2`) определен тип `const string &`, но фактические аргументы (`input` и `"***"`) имеют тип `string` и `const char *` соответственно. Поскольку аргумент `input` имеет тип `string`, ссылка переменной `s1` на него не вызывает затруднений. Но как программа воспримет передачу указателя на тип `char` в качестве аргумента ссылке на тип `string`?

Здесь имеют значение два момента. Первый момент в том, что класс `string` определяет преобразование типа `char *` в `string`, и это делает возможным инициализацию объекта `string` строкой в стиле C. Второй момент связан со свойством ссылочных формальных параметров `const`, которое обсуждалось ранее в этой главе. Предположим, что тип фактического аргумента не соответствует типу ссылочного параметра, но может быть преобразован в него. Тогда программа создает временную переменную необходимого типа, инициализирует ее преобразованным значением и передает ссылку на временную переменную. Ранее приводился пример, что параметр типа `const double &` может в подобной манере обрабатывать аргумент типа `int`. Аналогично параметр типа `const string &` может обрабатывать аргумент `char *` или `const char *`.

Положительный результат заключается в том, что формальный параметр типа `const string &` допускает использование объекта `string` или строки в стиле C в качестве фактического аргумента, передаваемого функции. Примером строки в стиле C может служить строковый литерал в кавычках, массив `char` с завершающим нулевым символом или переменная-указатель на `char`. Поэтому следующая строка кода работает нормально:

```
result = version1(input, "***");
```

Функция `version2()` не создает временную строку. Вместо этого она напрямую изменяет исходную строку:

```
const string & version2(string & s1, const string & s2) // имеет побочный эффект
{
    s1 = s2 + s1 + s2;
    // Возврат ссылки, переданной функции, безопасен
    return s1;
}
```

Эта функция может изменять значение `s1`, т.к. переменная `s1`, в отличие от `s2`, объявлена без `const`.

Поскольку `s1` является ссылкой на объект (`input`) в `main()`, возврат этой переменной в качестве ссылки вполне допустим. По той же причине строка:

```
result = version2(input, "###");
```

эквивалентна следующему коду:

```
version2(input, "###");    // input изменен непосредственно version2()
result = input;            // ссылка на s1 является ссылкой на input
```

Однако из-за того, что `s1` является ссылкой на `input`, вызов этой функции имеет побочный эффект, заключающийся в изменении также и `input`:

```
Your original string: It's not my fault.
Your string enhanced: ###It's not my fault.###
Your original string: ###It's not my fault.###
```

Таким образом, если исходная строка не должна изменяться, такое решение ошибочно.

Третья версия функции в листинге 8.7 служит напоминанием о том, как поступать нельзя:

```
const string & version3(string & s1, const string & s2) // неудачное решение
{
    string temp;
    temp = s2 + s1 + s2;
    // Возврат ссылки на локальную переменную небезопасен
    return temp;
}
```

Здесь присутствует серьезная ошибка — возврат ссылки на переменную, объявленную локально внутри функции `version3()`. Эта функция компилируется (с выводом предупреждения), но при попытке выполнения программы происходит сбой. Непосредственно ошибку вызывает следующая операция присваивания:

```
result = version3(input, "@@@");
```

Здесь производится попытка ссылки на память, которая больше не используется.

Еще один урок ООП: объекты, наследование и ссылки

Классы `ostream` и `ofstream` вскрывают интересное свойство ссылок. Как упоминалось в главе 6, объекты типа `ofstream` могут использовать методы `ostream`, позволяя файловому вводу-выводу применять те же формы, что и консольный ввод-вывод. Средство языка, позволяющее передавать возможности из одного класса в другой, называется *наследованием*. Оно подробно рассматривается в главе 13. Вкратце, `ostream` называется *базовым классом* (поскольку класс `ofstream` основан на нем), а `ofstream` — *производным классом* (т.к. он порожден от `ostream`). Производный класс наследует методы базового класса. Это означает, что объект `ofstream` может использовать функции базового класса, такие как методы форматирования `precision()` и `setf()`.

Другой аспект наследования состоит в том, что ссылка на базовый класс может указывать на объект производного класса, не требуя приведения типа. На практике это позволяет определять функцию, обладающую параметром-ссылкой на базовый класс. Эта функция может взаимодействовать с объектами базового класса, а также с производными объектами. Например, функция с параметром типа `ostream &` может принимать объект `ostream`, такой как `cout` или `ofstream`. Их можно объявлять в функции с одинаковым успехом.

Указанные аспекты демонстрируются в листинге 8.8, где одна и та же функция используется для записи данных в файл и отображения тех же данных на экране. Изменяется только аргумент, который передается вызываемой функции. Представленная программа рассчитывает фокусное расстояние объектива телескопа (его основного зеркала или линзы) и отдельных окуляров. Затем вычисляется кратность увеличения каждого окуляра телескопа. Кратность увеличения равна фокусному расстоянию объектива телескопа, деленному на фокусное расстояние используемого окуляра, так что вычисления здесь несложные. Кроме того, в программе используются некоторые методы форматирования, которые, как обещалось, одинаково успешно выполняются как с объектами типа `cout`, так и с объектами типа `ofstream` (в рассматриваемом примере — `fout`).

Листинг 8.8. filefunc.cpp

```
//filefunc.cpp -- функция с параметром ostream &
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
void file_it(ostream & os, double fo, const double fe[],int n);
const int LIMIT = 5;
int main()
{
   ofstream fout;
   const char * fn = "ep-data.txt";
   fout.open(fn);
   if (!fout.is_open())
   {
      cout << "Can't open " << fn << ".Bye.\n"; // не удается открыть файл
      exit(EXIT_FAILURE);
   }
   double objective;
   // Ввод фокусного расстояния объектива телескопа в мм
   cout << "Enter the focal length of your "
           "telescope objective in mm: ";
   cin >> objective;
   double eps[LIMIT];
   // Ввод фокусного расстояния окуляров в мм
   cout << "Enter the focal lengths, in mm, of " << LIMIT << " eyepieces:\n";
   for (int i = 0; i < LIMIT; i++)
   {
      cout << "Eyepiece #" << i + 1 << ": ";
      cin >> eps[i];
   }
   file_it(fout, objective, eps, LIMIT);
   file_it(cout, objective, eps, LIMIT);
   cout << "Done\n";
   return 0;
}
void file_it(ostream & os, double fo, const double fe[],int n)
{
   ios_base::fmtflags initial;
   initial = os.setf(ios_base::fixed); // сохранение исходного состояния форматирования
   os.precision(0);
   os << "Focal length of objective: " << fo << " mm\n"; // фокусное расстояние объектива
   os.setf(ios::showpoint);
   os.precision(1);
   os.width(12);
   os << "f.l. eyepiece";
   os.width(15);
   os << "magnification" << endl;     // коэффициент увеличения
   for (int i = 0; i < n; i++)
   {
      os.width(12);
      os << fe[i];
      os.width(15);
      os << int (fo/fe[i] + 0.5) << endl;
   }
   os.setf(initial);   // восстановление исходного состояния форматирования
}
```

Ниже показан пример выполнения программы из листинга 8.8:

```
Enter the focal length of your telescope objective in mm: 1800
Enter the focal lengths, in mm, of 5 eyepieces:
Eyepiece #1: 30
Eyepiece #2: 19
Eyepiece #3: 14
Eyepiece #4: 8.8
Eyepiece #5: 7.5
Focal length of objective: 1800 mm
f.l. eyepiece magnification
      30.0          60
      19.0          95
      14.0         129
       8.8         205
       7.5         240
Done
```

Следующая строка записывает данные окуляра в файл `ep-data.txt`:

```
file_it(fout, objective, eps, LIMIT);
```

А приведенная ниже строка выводит идентичную информацию в том же формате на экран:

```
file_it(cout, objective, eps, LIMIT);
```

Замечания по программе

Основная идея программы из листинга 8.8 состоит в демонстрации того факта, что параметр типа `ostream &` может ссылаться на объект `ostream`, такой как `cout`, и на объект `ofstream`, подобный `fout`. Кроме того, в программе также иллюстрируется использование методов форматирования объекта `ostream` для обоих типов параметров. (Более подробно эта тема рассматривается в главе 17.)

Метод `setf()` позволяет устанавливать различные состояния форматирования. Например, при вызове метода `setf(ios_base::fixed)` объект переводится в режим использования фиксированной десятичной точки.

При вызове метода `setf(ios_base::showpoint)` объект переводится в режим отображения завершающей десятичной точки, даже если последующие цифры являются нулями. Метод `precision()` указывает количество цифр, отображаемых справа от десятичной точки (если объект выводится в режиме `fixed`). Все эти установки сохраняются до тех пор, пока не будут изменены в результате следующего вызова метода. Вызов метода `width()` позволяет установить ширину поля для следующей операции вывода. Эта установка действует только для отображения единственного значения, а затем возвращается в принимаемое по умолчанию состояние. (По умолчанию ширина поля равна нулю. Затем ширина увеличивается до точного соответствия отображаемому значению.)

Функция `file_it()` содержит два интересных вызова методов:

```
ios_base::fmtflags initial;
initial = os.setf(ios_base::fixed); // сохранение исходного
                                    // состояния форматирования
...
os.setf(initial); // восстановление исходного состояния форматирования
```

Метод `setf()` возвращает копию всех настроек форматирования, которые действовали до его вызова. Обозначение `ios_base::fmtflags` является причудливым именем

типа данных, необходимых для хранения этой информации. В результате операции присваивания в переменной initial сохраняются настройки, которые действовали на момент вызова функции file_it(). Затем переменная initial может использоваться в качестве аргумента функции setf(), чтобы восстановить исходные значения всех установок форматирования. Таким образом, эта функция восстанавливает состояние объекта, которое существовало до его передачи функции file_it().

Более полное знакомство с классами поможет лучше уяснить работу этих методов, а также причину, по которой в программе часто используются обозначения вроде ios_base. Однако применение рассмотренных методов не обязательно откладывать до момента прочтения главы 17.

И последнее: каждый объект хранит собственные параметры форматирования. Поэтому, когда программа передает объект cout функции file_it(), его настройки форматирования изменяются, а затем восстанавливаются. То же самое происходит с объектом fout, когда он передается функции file_it().

Когда целесообразно использовать ссылочные аргументы

Есть две главных причины использовать ссылочные аргументы:

- чтобы позволить изменять объект данных в вызывающей функции;
- чтобы ускорить работу программы за счет передачи ссылки вместо полной копии объекта данных.

Вторая причина наиболее важна для крупных объектов данных, таких как структуры и объекты классов. По этим же двум причинам в качестве аргумента может использоваться указатель. Это оправдано, поскольку аргументы-ссылки, по сути, являются лишь альтернативным интерфейсом для кода, где применяются указатели. Итак, в каких случаях следует использовать ссылку, указатель или передачу по значению? Ниже приводятся основные рекомендации.

Функция использует передаваемые данные без их изменения в перечисленных ниже ситуациях.

- Если объект данных небольшой, например, такой как встроенный тип данных или некрупная структура, передавайте его по значению.
- Если объект данных представляет собой массив, используйте указатель, поскольку это единственный вариант. Объявите указатель с квалификатором const.
- Если объект данных является структурой приемлемого размера, используйте const-указатель или const-ссылку для увеличения эффективности программы. В этом случае удастся сохранить время и пространство, необходимое для копирования структуры или строения класса. Объявите указатель или ссылку с квалификатором const.
- Если объект данных является объектом класса, используйте ссылку с квалификатором const. Семантика строения класса часто требует применения ссылки. Эта главная причина добавления этого новшества в язык C++. Таким образом, стандартом является передача объектов класса по ссылке.

Функция изменяет данные вызывающей функции в следующих ситуациях.

- Если объект данных относится к одному из встроенных типов, используйте указатель. Если в коде встретилось выражение вида fixit(&x), где x имеет тип int, это явно означает, что функция должна изменять значение x.
- Если объект данных представляет собой массив, остается один выбор — указатель.

- Если объект данных является структурой, можно использовать ссылку или указатель.
- Когда объект данных представляет собой объект класса, следует применять ссылку.

Конечно, это лишь рекомендации, и могут существовать причины для других решений. Например, объект `cin` использует ссылки на базовые типы данных, поэтому вместо записи `cin >> &n` можно применять запись `cin >> n`.

Аргументы по умолчанию

Теперь рассмотрим еще одно новое инструментальное средство языка C++ — аргумент по умолчанию. *Аргумент по умолчанию* представляет собой значение, которое используется автоматически, если соответствующий фактический параметр в вызове функции не указан. Например, если функция `void wow(int n)` определена так, что `n` по умолчанию имеет значение `1`, то вызов функции `wow()` означает то же самое, что и `wow(1)`. Это свойство позволяет использовать функции более гибким образом. Предположим, что функция `left()` возвращает первые `n` символов строки, при этом сама строка и число `n` являются аргументами. Точнее, функция возвращает указатель на новую строку, представляющую собой выбранный фрагмент исходной строки.

Например, в результате вызова функции `left("theory", 3)` создается новая строка `"the"` и возвращается указатель на нее. Теперь предположим, что для второго аргумента установлено значение по умолчанию `1`. Вызов `left("theory", 3)` будет выполнен, как и раньше, поскольку указанная величина `3` переопределит значение по умолчанию. Однако вызов `left("theory")` теперь уже не будет ошибочным. Он подразумевает, что значение второго аргумента равно `1`, поэтому будет возвращен указатель на строку `"t"`. Этот вид выбора значений по умолчанию удобен для программ, которые часто извлекают строки длиной в один символ, но иногда требуется извлекать более длинные строки.

Как установить значение по умолчанию? Для этого применяется прототип функции. Поскольку компилятор использует прототип, чтобы узнать, сколько аргументов имеет функция, прототип функции также должен сообщить программе о возможности наличия аргументов по умолчанию. Метод заключается в присваивании значения аргументу в самом прототипе. Например, пусть имеется прототип, соответствующий следующему описанию функции `left()`:

```
char * left(const char * str, int n = 1);
```

Эта функция должна возвращать новую строку, поэтому ее типом будет `char*`, т.е. указатель на `char`. Если нужно сохранить исходную строку неизменной, следует использовать квалификатор `const` для первого аргумента. А чтобы аргумент `n` имел значение по умолчанию `1`, присвоим это значение аргументу `n`. Принимаемое по умолчанию значение аргумента — это значение, заданное при инициализации. Таким образом, данный прототип инициализирует `n` значением `1`. Если аргумент `n` опускается, для него принимается значение `1`, но если данный аргумент передается, то новое значение переопределяет значение `1`.

В функции со списком аргументов значения по умолчанию должны добавляться в конце. Другими словами, нельзя предоставить значение по умолчанию некоторому аргументу до тех пор, пока не будут предоставлены значения по умолчанию для всех аргументов, размещенных справа от него:

```
int harpo(int n, int m = 4, int j = 5);          // ПРАВИЛЬНО
int chico(int n, int m = 6, int j);              // НЕПРАВИЛЬНО
int groucho(int k = 1,  int m = 2,  int n = 3);  // ПРАВИЛЬНО
```

Например, прототип функции `harpo()` допускает реализацию вызова функции с одним, двумя или тремя аргументами:

```
beeps = harpo(2);          // то же, что и harpo(2,4,5)
beeps = harpo(1,8);        // то же, что и harpo(1,8,5)
beeps = harpo(8,7,6);      // аргументы по умолчанию не используются
```

Значения фактических аргументов присваиваются соответствующим формальным аргументам в направлении слева направо; пропускать аргументы нельзя. Таким образом, следующее выражение является недопустимым:

```
beeps = harpo(3, ,8);      // неправильно, m не устанавливается в 4
```

Аргументы по умолчанию не являются выдающимся достижением в программировании — они предназначены лишь для удобства. Когда вы начнете работать с классами, то убедитесь в том, что этот прием позволяет сократить количество конструкторов, методов и перегрузок методов, подлежащих определению.

Пример использования аргументов по умолчанию приведен в листинге 8.9. Обратите внимание, что значения по умолчанию отражает только прототип. Определение функции будет таким же, как и без аргументов по умолчанию.

Листинг 8.9. `left.cpp`

```
// left.cpp -- строковая функция с аргументом по умолчанию
#include <iostream>
const int ArSize = 80;
char * left(const char * str, int n = 1);
int main()
{
   using namespace std;
   char sample[ArSize];
   cout << "Enter a string:\n";
   cin.get(sample,ArSize);
   char *ps = left(sample, 4);
   cout << ps << endl;
   delete [] ps;          // освободить старую строку
   ps = left(sample);
   cout << ps << endl;
   delete [] ps;          // освободить новую строку
   return 0;
}
// Эта функция возвращает указатель на новую строку,
// состоящую из первых n символов строки str.
char * left(const char * str, int n)
{
    if(n < 0)
       n = 0;
   char * p = new char[n+1];
   int i;
   for (i = 0; i < n && str[i]; i++)
      p[i] = str[i];      // копирование символов
   while (i <= n)
      p[i++] = '\0';      // установка остальных символов строки в '\0'
   return p;
}
```

Ниже показан пример выполнения программы из листинга 8.9:

```
Enter a string:
forthcoming
fort
f
```

Замечания по программе

Программа из листинга 8.9 использует операцию `new` для создания новой строки, в которой будут храниться выбранные символы. Первое затруднение возникнет, когда пользователь укажет отрицательное количество символов. В этом случае функция устанавливает счетчик символов в `0` и в конечном итоге возвращает нулевую строку. Еще одно затруднение возникает, когда пользователь запрашивает количество символов, превышающее длину исходной строки. Функция защищена от подобных случаев за счет выполнения комбинированной проверки:

```
i < n && str[i]
```

Проверка `i < n` остановит цикл после того, как будут скопированы `n` символов. Вторая часть проверки — выражение `str[i]` — это код символа, копируемого в данный момент. Если цикл достигает нулевого символа, кодом которого является `0`, выполнение цикла прекратится. Заключительный цикл `while` завершает строку нулевым символом и заполняет остаток выделенного под строку пространства памяти нулевыми символами.

Другой способ установки размера новой строки состоит в том, чтобы присвоить переменной `n` меньшую величину из переданного значения и длины строки:

```
int len = strlen(str);
n = (n < len) ? n : len;    // меньшее из n и len
char * p = new char[n+1];
```

Это гарантирует, что `new` не выделит больше пространства, чем необходимо для хранения строки. Подобный подход полезен при наличии вызовов функции наподобие `left("Hi!", 32767)`. При первом подходе строка `"Hi!"` копируется в массив размером 32 767 символов, все элементы которого, за исключением первых трех, устанавливаются в нулевой символ. При втором подходе строка `"Hi!"` копируется в массив, состоящий из четырех символов. Но за счет добавления еще одного вызова функции (`strlen()`) размер программы увеличивается, ее выполнение замедляется и к тому же требуется включение заголовочного файла `cstring` (или `string.h`). Программисты на C предпочитают иметь более быстрый и компактный код, вследствие чего на них возлагается ответственность за правильное использование функций. Однако в C++ основное значение традиционно придается надежности. В конце концов, более медленная, зато правильно работающая программа лучше быстродействующей программе, которая работает некорректно. Если время, затраченное на вызов функции `strlen()`, неприемлемо, можно позволить непосредственно функции `left()` выбрать меньшее из значений `n` и длины строки. Например, приведенный ниже цикл завершается, когда `m` достигает значения `n` или длины строки, что бы ни случилось первым:

```
int m = 0;
while ( m <= n && str[m] != '\0')
   m++;
char * p = new char[m+1]:
// В остальном коде будет использоваться m вместо n
```

Вспомните, что выражение `str[m] != '\0'` вычисляется как `true`, когда `str[m]` не является нулевым символом, и как `false` — в противном случае. Поскольку в выражении `&&` ненулевые значения преобразуются в `true`, а нулевые — в `false`, проверочное условие `while` может быть также записано следующим образом:

```
while (m <= n && str[m])
```

Перегрузка функций

Полиморфизм функций — это удобное добавление C++ к возможностям языка C. В то время как аргументы по умолчанию позволяют вызывать одну и ту же функцию с различным количеством аргументов, *полиморфизм функций*, также называемый *перегрузкой функций*, предоставляет возможность использовать несколько функций с одним и тем же именем. Слово *полиморфизм* означает способность иметь множество форм, следовательно, полиморфизм функций позволяет функции иметь множество форм. Подобным же образом выражение *перегрузка функций* означает возможность привязки более чем одной функции к одному и тому же имени, таким образом, перегружая имя. Оба выражения означают одно и то же, но мы будем пользоваться вариантом *перегрузка функций*, как более строгим. С применением перегрузки функций можно разработать семейство функций, которые выполняют в точности одно и то же, но с использованием различных списков аргументов.

Перегруженные функции подобны глаголам, имеющим несколько значений. Например, глагол "болеть" в выражениях "Сергей болеет за местный футбольный клуб" и "Артем болеет уже третий день" имеет различный смысл. Контекст вам подскажет (надо надеяться), какое из значений подходит для того или иного случая. Аналогичным образом C++ использует контекст, чтобы решить, какой версией перегруженной функции следует воспользоваться.

Ключевую роль в перегрузке функций играет список аргументов, который также называется *сигнатурой функции*. Если две функции используют одно и то же количество аргументов с теми же самыми типами в одном и том же порядке, то функции имеют одинаковые сигнатуры; при этом имена переменных во внимание не принимаются. Язык C++ позволяет определить две функции с одним и тем же именем при условии, что эти функции обладают разными сигнатурами. Сигнатуры могут различаться по количеству аргументов или по их типам, либо по тому и другому. Например, можно определить набор функций `print()` со следующими прототипами:

```
void print(const char * str, int width);     // #1
void print(double d, int width);             // #2
void print(long l, int width);               // #3
void print(int i, int width);                // #4
void print(const char *str);                 // #5
```

При последующем вызове функции `print()` компилятор сопоставит используемый вариант с прототипом, имеющим ту же сигнатуру:

```
print("Pancakes", 15);                       // используется #1
print("Syrup");                              // используется #5
print(1999.0, 10);                           // используется #2
print(1999, 12);                             // используется #4
print(1999L, 15);                            // используется #3
```

Например, в вызове `print("Pancakes", 15)` передается строка и целое число в качестве аргументов, что соответствует прототипу #1.

При вызове перегруженных функций важно правильно указывать типы аргументов. Для примера рассмотрим следующие операторы:

```
unsigned int year = 3210;
print(year, 6);  // неоднозначный вызов
```

Какому прототипу соответствует этот вызов `print()`? Ни одному! Отсутствие подходящего прототипа не исключает автоматически использование одной из функций, поскольку C++ попытается выполнить стандартное приведение типов, чтобы достичь соответствия. Если бы, скажем, единственным прототипом функции `print()` был #2, вызов `print(year, 6)` повлек бы за собой преобразование значения `year` к типу `double`. Однако в приведенном выше коде имеется три прототипа с числом в качестве первого аргумента, при этом возникают три варианта преобразования аргумента `year`. В такой неоднозначной ситуации C++ отбрасывает подобный вызов функции как ошибочный.

Некоторые сигнатуры, которые выглядят как отличающиеся друг от друга, тем не менее, не могут сосуществовать. Для примера рассмотрим следующие два прототипа:

```
double cube(double x);
double cube(double & x);
```

Может показаться, что это именно тот случай, когда применима перегрузка функций, поскольку сигнатуры обеих функций вроде бы различны. Однако подойдем к этой ситуации с точки зрения компилятора. Предположим, что есть такой код:

```
cout << cube(x);
```

Аргумент `x` соответствует как прототипу `double x`, так и прототипу `double &x`. Следовательно, компилятор не может определить, какую функцию использовать. Во избежание такой путаницы, при проверке сигнатур функций компилятор считает, что ссылка на тип и сам тип являются одной и той же сигнатурой.

В ходе сопоставления функций учитываются различия между переменными с квалификатором `const` и без него. Рассмотрим следующие прототипы:

```
void dribble(char * bits);          // перегружена
void dribble (const char *cbits);   // перегружена
void dabble(char * bits);           // не перегружена
void drivel(const char * bits);     // не перегружена
```

Ниже приведены вызовы различных функций с указанием подходящих прототипов:

```
const char p1[20] = "How's the weather?";
char p2[20] = "How's business?";
dribble(p1);                        // dribble(const char *);
dribble(p2);                        // dribble(char *);
dabble(p1);                         // соответствия нет
dabble(p2);                         // dabble(char *);
drivel(p1);                         // drivel(const char *);
drivel(p2);                         // drivel(const char *);
```

Функция `dribble()` имеет два прототипа: один — для указателей `const` и один — для обычных указателей. Компилятор выбирает тот или другой прототип в зависимости от того, является ли фактический аргумент `const`. Функция `dabble()` соответствует только вызову с аргументом без `const`, а функция `drivel()` обеспечивает совпадение для вызовов с аргументами `const` или не `const`. Причина такого различия в поведении `drivel()` и `dabble()` связана с тем, что значение без `const` можно присвоить переменной `const`, но не наоборот.

Имейте в виду, что именно сигнатура, а не тип функции, делает возможным ее перегрузку. Например, два следующих объявления несовместимы:

```
long gronk(int n, float m);          // одинаковые сигнатуры, поэтому
double gronk(int n, float m);        // объявления не допускаются
```

В C++ нельзя перегружать функцию `gronk()` подобным образом. Можно иметь различные возвращаемые типы, но только при условии, что сигнатуры функций отличаются:

```
long gronk(int n, float m);          // различные сигнатуры, поэтому
double gronk(float n, float m);      // объявления допустимы
```

После обзора шаблонов позже в этой главе мы еще вернемся к теме соответствия функций.

Перегрузка ссылочных параметров

В классах и библиотеке STL часто используются ссылочные параметры, поэтому полезно знать, как перегрузка работает с различными ссылочными типами. Рассмотрим следующие три прототипа:

```
void sink(double & r1);              // соответствует изменяемому lvalue
void sank(const double & r2);        // соответствует изменяемому
                                     // или константному lvalue, rvalue
void sunk(double && r3);             // соответствует rvalue
```

Ссылочный параметр lvalue по имени `r1` соответствует изменяемому аргументу lvalue, такому как переменная `double`. Константный ссылочный параметр lvalue по имени `r2` соответствует изменяемому аргументу, константному аргументу lvalue и аргументу rvalue, такому как сумма двух значений `double`. И, наконец, ссылка rvalue по имени `r3` соответствует rvalue. Обратите внимание, что `r2` может соответствовать той же разновидности аргументов, как `r1` и `r3`. Возникает вопрос, а что случится, если перегрузить функцию с этими тремя типами параметров? Ответ заключается в том, что будет предпринят поиск более точного соответствия:

```
void staff(double & rs);             // соответствует изменяемому lvalue
voit staff(const double & rcs);      // соответствует rvalue,
                                     // константному lvalue
void stove(double & r1);             // соответствует изменяемому lvalue
void stove(const double & r2);       // соответствует константному lvalue
void stove(double && r3);            // соответствует rvalue
```

Это позволяет настраивать поведение функции на основе того, каковой является природа аргумента — lvalue, `const` или rvalue:

```
double x = 55.5;
const double y = 32.0;
stove(x);                            // вызывает stove(double &)
stove(y);                            // вызывает stove(const double &)
stove(x+y);                          // вызывает stove(double &&)
```

Если, скажем, опустить функцию `stove(double &&)`, то `stove(x+y)` взамен приведет к вызову `stove(const double &)`.

Пример перегрузки

Ранее в главе была создана функция `left()`, которая возвращает указатель на первые `n` символов в строке. Теперь разработаем еще одну функцию `left()`, но на этот раз она будет возвращать первые `n` цифр целочисленного значения. Она применима,

например, для анализа первых трех цифр почтового кода США, записанного в виде целое число, с целью сортировки по областям.

Целочисленную версию функции несколько труднее запрограммировать, чем строковую, поскольку цифры не хранятся в отдельных элементах массива. Один из способов решения предполагает сначала подсчет количества цифр в числе. Деление числа на 10 уменьшает его запись на одну цифру. Таким образом, можно воспользоваться делением, чтобы подсчитать количество цифр в числе. Точнее, эту процедуру можно выполнить в цикле, подобном приведенному ниже:

```
unsigned digits = 1;
while (n /= 10)
   digits++;
```

В этом цикле подсчитывается, сколько раз можно удалить цифру из числа `n` до того, как не останется ни одной цифры. Вспомните, что запись `n /= 10` является сокращением от `n = n / 10`. Если, например, `n` равно `8`, то при вычислении проверяемого выражения переменной `n` присваивается значение `8 / 10`, или `0`, поскольку деление целочисленное. Выполнение цикла при этом прекращается, и значение переменной `digits` остается равным `1`. Но если значение `n` равно `238`, то на первом этапе проверки условия цикла переменной `n` присваивается значение `238 / 10`, или `23`. Это значение не равно нулю, поэтому в ходе выполнения цикла значение переменной `digits` увеличивается до 2. На следующем этапе цикла значение `n` устанавливается равным `23 / 10`, или 2. Эта величина также не равна нулю, поэтому значение `digits` возрастает до 3. На следующем этапе цикла значение `n` устанавливается равным `2 / 10`, или 0, и выполнение цикла прекращается, а значение переменной `digits` остается равным `3`, что и является правильным результатом.

Теперь предположим, что исходное число содержит пять цифр и нужно возвратить первые три цифры. Чтобы получить данное трехзначное число, можно два раза разделить исходное число на 10. При каждом делении числа на 10 в записи числа удаляется одна цифра справа. Чтобы узнать, какое количество цифр нужно удалить, следует просто вычислить количество цифр, которые требуется отобразить, из общего количества цифр в представлении исходного числа. Например, чтобы отобразить четыре цифры числа, представленного девятью цифрами, удалите последние пять цифр. Это решение можно представить в виде следующего кода:

```
ct = digits - ct;
while (ct--)
   num /= 10;
return num;
```

В листинге 8.10 этот код помещен в новую функцию `left()`. Данная функция содержит и другие операторы, предназначенные для обработки специальных случаев, таких как запрос вывода нулевого количества цифр или количества, превышающего длину представления исходного числа. Поскольку сигнатура новой функции `left()` отличается от сигнатуры старой функции `left()`, мы получаем возможность использовать обе функции в одной и той же программе.

Листинг 8.10. `leftover.cpp`

```
// leftover.cpp -- перегрузка функции left()
#include <iostream>
unsigned long left(unsigned long num, unsigned ct);
char * left(const char * str, int n = 1);
```

```
int main()
{
   using namespace std;
   char * trip = "Hawaii!!";        // тестовое значение
   unsigned long n = 12345678;      // тестовое значение
   int i;
   char * temp;
   for (i = 1; i < 10; i++)
   {
      cout << left(n, i) << endl;
      temp = left(trip,i);
      cout << temp << endl;
      delete [] temp;               // указатель на временную область хранения
   }
   return 0;
}
// Возвращает первых ct цифр числа num
unsigned long left(unsigned long num, unsigned ct)
{
   unsigned digits = 1;
   unsigned long n = num;
   if (ct == 0 || num == 0)
      return 0;                     // возврат 0 в случае отсутствия цифр
   while (n /= 10)
      digits++;
   if (digits > ct)
   {
      ct = digits - ct;
      while (ct--)
         num /= 10;
      return num;                   // возврат ct знаков слева
   }
   else  // если ct >= количества цифр
      return num;                   // возврат числа целиком
}
// Возвращает указатель на новую строку, состоящую
// из n первых символов строки str
char * left(const char * str, int n)
{
   if(n < 0)
   n = 0;
   char * p = new char[n+1];
   int i;
   for (i = 0; i < n && str[i]; i++)
      p[i] = str[i];                // копирование символов
   while (i <= n)
      p[i++] = '\0';                // установка остальных символов строки в '\0'
   return p;
}
```

Ниже показан вывод программы из листинга 8.10:

```
1
H
12
Ha
123
```

```
Haw
1234
Hawa
12345
Hawai
123456
Hawaii
1234567
Hawaii!
12345678
Hawaii!!
12345678
Hawaii!!
```

Когда целесообразно использовать перегрузку функций

Возможность перегрузки функций может произвести большое впечатление, но злоупотреблять ею не следует. Перегрузку целесообразно использовать для функций, которые выполняют в основном одни и те же действия, но с различными типами данных. Кроме того, имеет смысл оценить возможность достижения той же цели посредством аргументов, принимаемых по умолчанию. Например, можно заменить единственную функцию `left()`, предназначенную для обработки строк, двумя перегруженными функциями:

```
char * left(const char * str, unsigned n);        // два аргумента
char * left(const char * str);                    // один аргумент
```

Тем не менее, использование единственной функции с аргументом по умолчанию будет проще. Прежде всего, понадобится написать только одну функцию, а не две, к тому же программе потребуется меньше памяти. Если впоследствии нужно будет внести изменения, достаточно отредактировать только одну функцию. Однако если необходимо применять аргументы разного типа, то аргументы по умолчанию здесь не помогут, и придется использовать перегрузку функций.

Что такое декорирование имен?

Как в C++ различаются перегруженные функции? Каждой из таких функций назначается скрытый идентификатор. Когда вы используете редактор среды разработки на C++ и компилируете программы, компилятор C++ выполняет то, что называется *декорированием имен* или *искажением имен*; при этом имя каждой функции шифруется на основе типов формальных параметров, указанных в прототипе функции. Рассмотрим следующий прототип в недекорированном виде:

```
long MyFunctionFoo(int, float);
```

Этот формат удобен для восприятия человеком; мы видим, что функция принимает два аргумента с типами `int` и `float`, а возвращает значение типа `long`. Для собственных нужд компилятор документирует этот интерфейс, преобразуя имя во внутреннее представление с более сложным внешним видом, которое имеет примерно следующий вид:

```
?MyFunctionFoo@@YAXH
```

В этом невразумительном обозначении закодировано количество аргументов и их типы. Получаемый набор символов зависит от сигнатуры функции, а также от применяемого компилятора.

Шаблоны функций

Современные компиляторы C++ реализуют одно из новейших добавлений к языку: шаблоны функций. *Шаблон функции* – это обобщенное описание функции; т.е. он определяет функцию в терминах обобщенного типа, вместо которого может быть подставлен определенный тип данных, такой как `int` или `double`. Передавая шаблону тип в качестве параметра, можно заставить компилятор сгенерировать функцию для этого конкретного типа. Поскольку шаблоны позволяют программировать в терминах обобщенного, а не специфического типа, этот процесс иногда называют *обобщенным программированием*. Поскольку типы представлены параметрами, средство шаблонов иногда называют *параметризированными типами*. Давайте посмотрим, чем это средство полезно, и как оно работает.

Ранее в листинге 8.4 была определена функция, которая осуществляет обмен значениями двух переменных `int`. Предположим, что вместо этого необходимо произвести обмен значениями двух переменных `double`. Один из подходов к решению этой задачи состоит в дублировании исходного программного кода с последующей заменой каждого слова `int` словом `double`. Чтобы произвести обмен значениями двух переменных `char`, эту процедуру придется повторить. Тем не менее, чтобы выполнить такие незначительные изменения, вам придется затратить время, которого всегда не хватает, при этом еще и не исключена вероятность ошибки. Если вносить эти изменения вручную, можно где-то пропустить `int`. Если же воспользоваться функцией глобального поиска и замены, например, слова `int` словом `double`, то строки

```
int x;
short interval;
```

можно превратить в следующие:

```
double x;                 // намеренное изменение типа
short doubleerval;        // непреднамеренное изменение имени переменной
```

Средство шаблонов функций C++ автоматизирует этот процесс, обеспечивая высокую надежность и экономию времени.

Шаблоны функций позволяют определять функции в терминах некоторого произвольного типа. Например, можно создать шаблон для осуществления обмена значениями, подобный приведенному ниже:

```
template <typename AnyType>
void Swap(AnyType &a, AnyType &b)
{
   AnyType temp;
   temp = a;
   a = b;
   b = temp;
}
```

Первая строка указывает, что устанавливается шаблон, а произвольный тип данных получает имя `AnyType`. Ключевые слова `template` и `typename` являются обязательными; при этом вместо `typename` можно использовать ключевое слово `class`. Кроме того, должны присутствовать угловые скобки. Имя типа может быть любым (в этом примере – `AnyType`), при условии, что оно соответствует обычным правилам именования C++; многие программисты используют простые имена, такие как `T`. Остальная часть кода описывает алгоритм обмена значениями типа `AnyType`. Никаких функций шаблон не создает. Вместо этого он предоставляет компилятору указания по определе-

нию функции. Если необходима функция для обмена значениями int, компилятор создаст функцию согласно этому шаблону, подставляя int вместо AnyType. Аналогично, когда нужна функция для обмена значениями double, компилятор будет руководствоваться шаблоном, подставляя тип double вместо AnyType.

Перед тем, как в стандарте C++98 было добавлено новое ключевое слово typename, в рассматриваемом контексте использовалось ключевое слово class. Это значит, что определение шаблона можно записать в следующей форме:

```
template <class AnyType>
void Swap(AnyType &a, AnyType &b)
{
   AnyType temp;
   temp = a;
   a = b;
   b = temp;
}
```

Ключевое слово typename делает более очевидным тот факт, что параметр AnyType представляет тип; однако к тому времени были созданы большие библиотеки кода, в которых применялось старое ключевое слово class. В приведенном контексте стандарт C++ трактует оба эти ключевых слова как идентичные. В этой книге используются обе формы.

Совет

Шаблоны должны использоваться в тех случаях, когда необходимы функции, применяющие один и тот же алгоритм к различным типам данных. Если перед вами не стоит задача обеспечения обратной совместимости и не затрудняет набор более длинного слова, можете использовать при объявлении параметров типа ключевое слово typename, а не class.

Чтобы сообщить компилятору о том, что нужна определенная форма функции обмена значениями, в программе достаточно вызвать функцию Swap(). Компилятор проанализирует типы передаваемых аргументов, а затем сгенерирует соответствующую функцию. В листинге 8.11 показано, как это делается. Формат программы выбран по образцу для обычных функций — прототип шаблона функции располагается в верхней части файла, а определение шаблона функции следует сразу после main().

Листинг 8.11. funtemp.cpp

```
// funtemp.cpp — использование шаблона функции
#include <iostream>
// Прототип шаблона функции
template <typename T>          // или class T
void Swap(T &a, T &b);

int main()
{
   using namespace std;
   int i = 10;
   int j = 20;
   cout << "i, j = " << i << ", " << j << ".\n";
   cout << "Using compiler-generated int swapper:\n";
   Swap(i,j);                 // генерирует void Swap(int &, int &)
   cout << "Now i, j = " << i << ", " << j << ".\n";
   double x = 24.5;
   double y = 81.7;
   cout << "x, y = " << x << ", " << y << ".\n";
   cout << "Using compiler-generated double swapper:\n";
```

```
    Swap(x,y);              // генерирует void Swap(double &, double &)
    cout << "Now x, y = " << x << ", " << y << ".\n";
    // cin.get();
    return 0;
}
// Определение шаблона функции
template <typename T>     // или class T
void Swap(T &a, T &b)
{
    T temp;               // temp - переменная типа T
    temp = a;
    a = b;
    b = temp;
}
```

Первая функция `Swap()` в листинге 8.11 имеет два аргумента типа `int`, поэтому компилятор генерирует версию функции, предназначенную для обработки данных типа `int`. Другими словами, он заменяет каждое использование `T` типом `int`, создавая определение следующего вида:

```
void Swap(int &a, int &b)
{
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Вы не увидите этот код, но компилятор генерирует его, а затем использует в программе. Вторая функция `Swap()` имеет два аргумента типа `double`, поэтому компилятор генерирует версию функции, предназначенную для обработки данных `double`. Таким образом, он заменяет все вхождения `T` типом `double`, генерируя следующий код:

```
void Swap(double &a, double &b)
{
    double temp;
    temp = a;
    a = b;
    b = temp;
}
```

Вывод программы из листинга 8.11 показывает, что все работает, как и ожидалось:

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
x, y = 24.5, 81.7.
Using compiler-generated double swapper:
Now x, y = 81.7, 24.5.
```

Обратите внимание, что шаблоны функций не сокращают размеры исполняемых файлов. В листинге 8.11 все по-прежнему завершается двумя отдельными определениями функций, как если бы это было реализовано вручную. Окончательный код не содержит шаблонов, а содержит реальные функции, сгенерированные для программы. Преимущество шаблонов состоит в упрощении процесса генерации нескольких определений функции, а также в увеличении его надежности.

Чаще всего шаблоны помещаются в заголовочный файл, который затем включается в использующий эти шаблоны файл. Заголовочные файлы обсуждаются в главе 9.

Перегруженные шаблоны

Шаблоны используются, когда необходимо создать функции, которые применяют один и тот же алгоритм к различным типам, как было показано в листинге 8.11. Однако, возможно, не для всех типов этот алгоритм выглядит совершенно одинаково. В таких случаях можно перегрузить определения шаблонов, точно так же, как перегружаются обычные функции. Как и при перегрузке функций, перегруженные шаблоны должны иметь различные сигнатуры. Например, в листинг 8.12 добавлен новый шаблон для обмена элементами между двумя массивами. Исходный шаблон имеет сигнатуру `(T &, T &)`, в то время как новый шаблон – сигнатуру `(T [], T [], int)`. Обратите внимание, что последним аргументом является конкретный тип (`int`), а не обобщенный. Не все аргументы шаблонов обязательно должны иметь обобщенный тип.

Когда в файле `twotemps.cpp` компилятор встречает первый вызов `Swap()`, он обнаруживает, что в нем имеется два аргумента типа `int`, и сопоставляет его с исходным шаблоном. Однако во втором случае использования этой функции в качестве аргументов выступают два массива `int` и значение `int`, что соответствует новому шаблону.

Листинг 8.12. `twotemps.cpp`

```
// twotemps.cpp -- использование перегруженных шаблонов функций
#include <iostream>
template <typename T>           // исходный шаблон
void Swap(T &a, T &b);

template <typename T>           // новый шаблон
void Swap(T *a, T *b, int n);
void Show(int a[]);
const int Lim = 8;

int main()
{
    using namespace std;
    int i = 10, j = 20;
    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";
    Swap(i,j);                  // соответствует исходному шаблону
    cout << "Now i, j = " << i << ", " << j << ".\n";
    int d1[Lim] = {0,7,0,4,1,7,7,6};
    int d2[Lim] = {0,7,2,0,1,9,6,9};
    cout << "Original arrays:\n";
    Show(d1);
    Show(d2);
    Swap(d1,d2,Lim);            // соответствует новому шаблону
    cout << "Swapped arrays:\n";
    Show(d1);
    Show(d2);
    // cin.get();
    return 0;
}

template <typename T>
void Swap(T &a, T &b)
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

```
template <typename T>
void Swap(T a[], T b[], int n)
{
    T temp;
    for (int i = 0; i < n; i++)
    {
        temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}
void Show(int a[])
{
    using namespace std;
    cout << a[0] << a[1] << "/";
    cout << a[2] << a[3] << "/";
    for (int i = 4; i < Lim; i++)
        cout << a[i];
    cout << endl;
}
```

Ниже показаны результаты выполнения программы из листинга 8.12:

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Original arrays:
07/04/1776
07/20/1969
Swapped arrays:
07/20/1969
07/04/1776
```

Ограничения шаблонов

Предположим, что имеется следующий шаблон функции:

```
template <class T> // или template <typename T>
void f(T a, T b)
{...}
```

Часто в коде делаются предположения относительно того, как операции возможны для того или иного типа. Например, в следующем операторе предполагается, что определена операция присваивания, но это не будет верно, если типом `T` является встроенный массив:

```
a = b;
```

Подобным же образом, ниже предполагается, что определена операция >, что не будет справедливо, если типом `T` окажется обычная структура:

```
if (a > b)
```

Также хотя операция > определена для имен массивов, поскольку они являются адресами, данная операция сравнивает адреса массивов, а это может быть не тем, что имело в виду. В приведенном ниже операторе предполагается, что для типа `T` определена операция умножения, а это не так в случае, когда `T` — массив, указатель или структура:

```
T c = a*b;
```

Словом, довольно легко получить шаблон функции, который не может обрабатывать определенные типы. С другой стороны, иногда обобщение имеет смысл, даже если обычный синтаксис C++ не допускает его. Например, сложение структур, содержащих координаты позиции, вполне оправдано, несмотря на то, что операция + для структур не определена. Один подход связан с тем, что C++ позволяет перегружать операцию +, так что она может быть использована с отдельной формой структуры или класса. Такая возможность обсуждается в главе 11. Шаблон, который требует использования операции +, затем сможет поддерживать структуру, имеющую перегруженную операцию +. Другой подход заключается в предоставлении специализированных определений шаблонов для отдельных типов. Давайте взглянем на это в следующем разделе.

Явные специализации

Предположим, что имеется следующее определение структуры:

```
struct job
{
   char name[40];
   double salary;
   int floor;
};
```

Также предположим, что необходимо обеспечить возможность обмена содержимым этих структур. Исходный шаблон использует следующий код для выполнения обмена:

```
temp = a;
a = b;
b = temp;
```

Поскольку в C++ разрешено присваивать одну структуру другой, этот код работает безупречно даже в том случае, когда тип `T` является структурой `job`. Однако предположим, что требуется совершить обмен данными только между членами `salary` и `floor`, а члены `name` оставить без изменений. Это требует другого кода, но аргументы функции `Swap()` будут такими же, как и в первом случае (ссылки на две структуры `job`), так что применить перегрузку шаблона для предоставления альтернативного кода не удастся.

Однако можно предоставить специализированное определение функции, называемое *явной специализацией*, которое содержит требуемый код. Если компилятор обнаруживает специализированное определение, которое точно соответствует вызову функции, он использует его без поиска шаблонов. Механизм специализации претерпевал изменения по мере эволюции языка. Мы рассмотрим форму, действующую на текущий момент и регламентируемую стандартом C++.

Специализация третьего поколения (стандарт ISO/ANSI C++)

После ряда ранних экспериментов с другими подходами, в стандарте C++98 принят следующий подход.

- Одно и то же имя может применяться для нешаблонной функции, шаблонной функции и явной специализации шаблона, а также всех перегруженных версий всего перечисленного.
- Прототип и определение явной специализации должно быть предварено `template <>`, а также указывать имя обобщенного типа данных.
- Специализация переопределяет обычный шаблон, а нешаблонная функция переопределяет и специализацию, и шаблон.

Ниже приведены примеры прототипов всех трех форм для обмена структур типа job.

```
// Прототип нешаблонной функции
void Swap(job &, job &);
// Прототип шаблона
template <typename T>
void Swap(T &, T &);
// Явная специализация для типа job
template <> void Swap<job>(job &, job &);
```

Как уже упоминалось ранее, если существует более одного из перечисленных прототипов, компилятор отдает предпочтение нешаблонной версии перед явными специализациями и шаблонными версиями, и предпочитает явную специализацию перед версией, сгенерированной из шаблона. В следующем примере первый вызов функции Swap() использует обобщенный шаблон, а второй вызов — явную специализацию, основанную на типе job:

```
...
template <class T>              // шаблон
void Swap(T &, T &);
// Явная специализация для типа job
template <> void Swap<job>(job &, job &);
int main()
{
   double u, v;
   ...
   Swap(u,v);                   // используется шаблон
   int a, b;
   ...
   Swap(a,b);                   // используется void Swap<job>(job &, job &)
}
```

Конструкция <job> в выражении Swap<job> необязательна, поскольку типы аргументов функции указывают, что это специализация для структуры job. Поэтому прототип может иметь и такой вид:

```
template <> void Swap (job &, job &);    // упрощенная форма
```

В старых версиях компиляторов используется правила специализации, предшествующие стандарту C++, но давайте сначала посмотрим, как должны работать явные специализации.

Пример явной специализации

Программа, представленная в листинге 8.13, иллюстрирует работу явной специализации.

Листинг 8.13. twoswap.cpp

```
// twoswap.cpp -- специализация переопределяет шаблон
#include <iostream>
template <typename T>
void Swap(T &a, T &b);
struct job
{
   char name[40];
   double salary;
   int floor;
};
```

```
// Явная специализация
template <> void Swap<job>(job &j1, job &j2);
void Show(job &j);

int main()
{
    using namespace std;
    cout.precision(2);
    cout.setf(ios::fixed, ios::floatfield);
    int i = 10, j = 20;

    cout << "i, j = " << i << ", " << j << ".\n";
    cout << "Using compiler-generated int swapper:\n";

    Swap(i,j);                          // генерирует void Swap(int &, int &)
    cout << "Now i, j = " << i << ", " << j << ".\n";
    job sue = {"Susan Yaffee", 73000.60, 7};
    job sidney = {"Sidney Taffee", 78060.72, 9};
    cout << "Before job swapping:\n";

    Show(sue);
    Show(sidney);
    Swap(sue, sidney);                  // использует void Swap(job &, job &)
    cout << "After job swapping:\n";

    Show(sue);
    Show(sidney);
    // cin.get();
    return 0;
}

template <typename T>
void Swap(T &a, T &b)                   // обобщенная версия
{
    T temp;
    temp = a;
    a = b;
    b = temp;
}

// Обменивается только содержимым полей salary и floor структуры job
template <> void Swap<job>(job &j1, job &j2) // специализация
{
    double t1;
    int t2;
    t1 = j1.salary;
    j1.salary = j2.salary;
    j2.salary = t1;
    t2 = j1.floor;
    j1.floor = j2.floor;
    j2.floor = t2;
}

void Show(job &j)
{
    using namespace std;
    cout << j.name << ": $" << j.salary
         << " on floor " << j.floor << endl;
}
```

Ниже показан вывод программы из листинга 8.13:

```
i, j = 10, 20.
Using compiler-generated int swapper:
Now i, j = 20, 10.
Before job swapping:
Susan Yaffee: $73000.60 on floor 7
Sidney Taffee: $78060.72 on floor 9
After job swapping:
Susan Yaffee: $78060.72 on floor 9
Sidney Taffee: $73000.60 on floor 7
```

Создание экземпляров и специализация

Чтобы расширить понимание шаблонов, необходимо ознакомиться с понятиями *создание экземпляров* и *специализация*. Имейте в виду, что включение шаблона функции в код само по себе не приводит к генерации определения функции. Это просто план для формирования определения функции. Когда компилятор использует шаблон при генерации определения функции для определенного типа данных, результат называется *созданием экземпляра* шаблона. Например, в листинге 8.13 вызов функции `Swap(i,j)` заставляет компилятор сгенерировать экземпляр `Swap()`, используя `int` в качестве типа. Шаблон — это *не* определение функции, но определенный экземпляр шаблона, использующий `int`, *является* определением функции. Такой вид создания экземпляров шаблонов называется *неявным созданием экземпляров*, поскольку компилятор выясняет необходимость в построении определения, обнаруживая тот факт, что в программе используется функция `Swap()` с параметрами `int`.

Первоначально неявное создание экземпляров было единственным способом, посредством которого компилятор генерировал определения функций из шаблонов, однако сейчас C++ позволяет выполнять *явное создание экземпляров*. Это означает возможность дать компилятору прямую команду создать определенный экземпляр, например, `Swap<int>()`. Синтаксис предусматривает объявление с использованием нотации <> для указания типа и предварение объявления ключевым словом `template`:

```
template void Swap<int>(int, int);      // явное создание экземпляра
```

Компилятор, в котором реализована эта возможность, обнаружив такое объявление, использует шаблон функции `Swap()`, чтобы сгенерировать экземпляр функции с типом `int`. То есть такое объявление означает "использовать шаблон функции `Swap()`, чтобы сгенерировать определение функции для типа `int`". Сравните явное создание экземпляров с явной специализацией, которая применяет одно из следующих эквивалентных объявлений:

```
template <> Swap<int>(int &, int &);    // явная специализация
template <> Swap(int &, int &);         // явная специализация
```

Различие состоит в том, что эти два объявления означают следующее: "не применять шаблон функции `Swap()`, чтобы сгенерировать определение функции; вместо этого воспользоваться отдельным специализированным определением функции, явно сформулированным для типа `int`". Эти прототипы должны быть ассоциированы с собственными определениями функций. В объявлении явной специализации после ключевого слова `template` следует конструкция <>. В объявлении явного создания экземпляра она опускается.

Внимание!

Попытка одновременного использования в одном файле или, в более общем случае — в компилируемом модуле, как явного создания экземпляра, так и явной специализации для одного и того же типа (типов) приведет к ошибке.

Явные создания экземпляров также могут быть обеспечены за счет использования функции в программе. Например, взгляните на следующий код:

```
template <class T>
T Add(T a, T b)                          // передача по значению
{
   return a + b;
}
...
int m = 6;
double x = 10.2;
cout << Add<double>(x, m) << endl;       // явное создание экземпляра
```

Этот шаблон не даст соответствия с вызовом функции `Add(x, m)`, поскольку шаблон ожидает, что оба аргумента функции относятся к одному и тому же типу. Но использование `Add<double>(x, m)` приводит к созданию экземпляра для типа `double`, и тип аргумента `m` приводится к `double` для соответствия второму параметру функции `Add<double>(double, double)`.

А что если поступить похожим образом с функцией `Swap()`?

```
int m = 5;
double x = 14.3;
Swap<double>(m, x);  // почти работает
```

Это явно создаст экземпляр для типа `double`. К сожалению, в данном случае код работать не будет, т.к. первый формальный параметр, имея тип `double &`, не может ссылаться на переменную `m` типа `int`.

Неявное и явное создание экземпляров, а также явная специализация, вместе называются *специализацией*. Общим для них является то, что они представляют определение функции, в основу которого положены специфические типы данных, а не определение функции, являющееся обобщенным описанием.

Добавление явного создания экземпляров привело к появлению нового синтаксиса префиксов `template` и `template <>` в объявлениях, что дает возможность провести различие между явным созданием экземпляров и явной специализацией. Чаще всего бывает так, что более широкие возможности обусловливаются увеличением количества синтаксических правил. Следующий фрагмент кода служит сводкой рассмотренных концепций:

```
...
template <class T>
void Swap (T &, T &);                          // прототип шаблона
template <> void Swap<int>(job &, job &); // явная специализация для job
int main(void)
{
  template void Swap<char>(char &, char &);  // явное создание экземпляра для char
  short a, b;
  ...
  Swap(a,b);   // неявное создание экземпляра шаблона для short
  job n, m;
  ...
  Swap(n, m);  // использование явной специализации для job
  char g, h;
  ...
  Swap(g, h);  // использование явного создания экземпляра шаблона для char
  ...
}
```

Когда компилятор обнаруживает явное создание экземпляра для `char`, он использует определение шаблона, чтобы сгенерировать версию функции `Swap()`, предназначенную для типа `char`. В остальных вызовах `Swap()` компилятор сопоставляет шаблон с используемыми в вызове фактическими аргументами. Например, когда компилятор обнаруживает вызов функции `Swap(a,b)`, он генерирует версию этой функции для типа `short`, поскольку оба аргумента принадлежат этому типу. Когда компилятор обнаруживает вызов функции `Swap(n,m)`, он использует отдельное определение (явную специализацию), предоставленное для типа `job`. Когда компилятор достигает вызова функции `Swap(g,h)`, он применяет специализацию шаблона, которая уже была сгенерирована во время обработки явного создания экземпляра.

Какую версию функции выбирает компилятор?

В отношении перегрузки функций, шаблонов функций и перегрузки шаблонов функций язык C++ располагает четко определенной стратегией выбора определения функции, применяемого для данного вызова функции, особенно при наличии множества аргументов. Этот процесс выбора называется *разрешением перегрузки*. Детальное описание стратегии требует отдельной главы, поэтому здесь мы рассмотрим работу этого процесса в общих чертах.

- **Фаза 1.** Составьте список функций-кандидатов. Таковыми являются функции и шаблоны функций с таким же именем, как у вызываемой функции.
- **Фаза 2.** Беря за основу список функций-кандидатов, составьте список подходящих функций. Таковыми являются функции с корректным количеством аргументов, для которых существует неявная последовательность преобразований типов. Она включает случай точного совпадения типа каждого фактического аргумента с типом соответствующего формального аргумента. Например, при вызове функции с аргументом типа `float` это значение может быть приведено к типу `double` для соответствия типу `double` формального параметра, а шаблон может сгенерировать экземпляр функции для типа `float`.
- **Фаза 3.** Проверьте наличие наиболее подходящей функции. Если она есть, используйте ее. В противном случае вызов функции является ошибочным.

Рассмотрим пример вызова функции с единственным аргументом:

```
may('B');  // фактический аргумент имеет тип char
```

Прежде всего, компилятор отмечает все кандидаты, каковыми являются функции и шаблоны функций с именем `may()`. Затем он находит среди них те, которые могут быть вызваны с одним аргументом. Например, в этом случае проверку пройдут следующие функции, поскольку они имеют одно и то же имя и могут использоваться с одним аргументом:

```
void may(int);                          // #1
float may(float, float = 3);            // #2
void may(char);                         // #3
char * may(const char *);               // #4
char may(const char &);                 // #5
template<class T> void may(const T &);  // #6
template<class T> void may(T *);        // #7
```

Обратите внимание, что при этом учитываются только сигнатуры, а не типы возвращаемых значений. Однако два кандидата (#4 и #7) из списка не подходят, поскольку целочисленный тип данных не может быть преобразован неявно (т.е. без явного

приведения типов) в тип указателя. Оставшийся шаблон подходит, т.к. может быть использован для генерирования специализации, где в качестве T принимается тип char. В итоге остается пять функций-кандидатов, каждая из которых может использоваться так, как если бы она была единственной объявленной функцией.

Далее компилятор должен определить, какая из функций-кандидатов в наибольшей степени соответствует критерию отбора. Он анализирует преобразования, необходимые для того, чтобы аргумент обращения к функции соответствовал аргументу наиболее подходящего кандидата. В общем случае порядок следования от наилучшего к наихудшему варианту можно представить следующим образом.

1. Точное соответствие, при этом обычные функции имеют приоритет перед шаблонами.
2. Преобразование за счет расширения (например, автоматические преобразования char и short в int и float в double).
3. Преобразование с помощью стандартных преобразований (например, преобразование int в char или long в double).
4. Преобразования, определяемые пользователем, такие как те, что определены в объявлениях классов.

Например, функция #1 предпочтительнее функции #2, поскольку преобразование char в int является расширением (см. главу 3), в то время как char в float — это стандартное преобразование (также описанное в главе 3). Функции #3, #5 и #6 предпочтительнее функций #1 и #2, т.к. они являются точными соответствиями. Функции #3 и #5 предпочтительнее варианта #6, потому что последний представляет собой шаблон. Этот анализ порождает пару вопросов. Что такое точное соответствие? Что произойдет, если таких соответствий будет два, как в случае функций #3 и #5? Обычно, как и в рассматриваемом примере, два точных соответствия приводят к ошибке, но из этого правила существуют исключения. Очевидно, этот вопрос требует дополнительного изучения.

Точные соответствия и наилучшие соответствия

При достижении точного соответствия C++ допускает некоторые "тривиальные преобразования". Список таких преобразований представлен в табл. 8.1; здесь с помощью Type обозначается произвольный тип данных.

Таблица 8.1. Тривиальные преобразования, допустимые при точном соответствии

Из фактического аргумента	В формальный аргумент
`Type`	`Type &`
`Type &`	`Type`
`Type []`	`* Type`
`Type (`*список-аргументов*`)`	`Type (*)(`*список-аргументов*`)`
`Type`	`const Type`
`Type`	`volatile Type`
`Type *`	`const Type *`
`Type *`	`volatile Type *`

Например, фактический аргумент `int` является точным соответствием формальному параметру `int &`. Обратите внимание, что `Type` может быть чем-то подобным `char &`, так что эти правила включают преобразование `char &` в `const char &`. Запись `Type (список-аргументов)` означает, что имя функции как фактический аргумент соответствует указателю на функцию, переданному в качестве формального параметра, при условии, что оба они имеют один и тот же возвращаемый тип и список аргументов. (Указатели на функции обсуждались в главе 7. Там же рассматривалась возможность передачи имени функции в качестве аргумента функции, которая ожидает указателя на функцию.) Ключевое слово `volatile` рассматривается в главе 9.

Предположим, что есть следующий код функции:

```
struct blot { int a; char b[10]; };
blot ink = { 25, "spots" };
...
recycle(ink);
```

В этом случае все перечисленные ниже прототипы будут точными соответствиями:

```
void recycle(blot);                 // #1  blot в blot
void recycle(const blot);           // #2  blot в const blot
void recycle(blot &);               // #3  blot в blot &
void recycle(const blot &);         // #4  blot в const blot &
```

Как и можно было предположить, результатом наличия множества подходящих прототипов является то, что компилятор не в состоянии завершить процесс разрешения перегрузки. Наиболее подходящей функции не существует, и компилятор сгенерирует сообщение об ошибке, в котором вероятно будет присутствовать слово "ambiguous" (неоднозначный).

Однако разрешение перегрузки иногда возможно даже в случае, когда две функции являются точным соответствием. Прежде всего, указатели и ссылки на данные не `const` сопоставляются преимущественно с указателями не `const` и ссылочными параметрами. То есть, если бы в примере с `recycle()` существовали только функции #3 и #4, то был бы выбран вариант #3, поскольку переменная `ink` не объявлена как `const`. Тем не менее, такое различение между `const` и не `const` применимо только к данным, на которые имеются ссылки и указатели. Другими словами, если бы доступными были только функции #1 и #2, то возникла бы ошибка, связанная с неопределенностью.

Другой случай, при котором одно точное соответствие оказывается лучше другого, касается ситуации, когда одна функция является нешаблонной, а другая — нет. Тогда нешаблонная функция рассматривается как более подходящая, чем шаблон, включая явные специализации.

Если, в конечном счете, оказалось два точных соответствия, и оба представляют собой шаблонные функции, то предпочитаемым вариантом будет шаблонная функция, являющаяся более специализированной (при наличии таковой). Это означает, например, что явная специализация получает преимущество перед функцией, неявно сгенерированной из шаблона:

```
struct blot { int a; char b[10]; };
template <class Type> void recycle (Type t);      // шаблон
template <> void recycle<blot> (blot & t);        // специализация для blot
...
blot ink = { 25, "spots" };
...
recycle(ink);  // используется специализация
```

Понятие *наиболее специализированная* не всегда означает явную специализацию; в принципе, оно отражает то, что при выборе компилятором используемого типа выполняется меньшее количество преобразований. В качестве примера рассмотрим два следующих шаблона:

```
template <class Type> void recycle (Type t);      // #1
template <class Type> void recycle (Type * t);    // #2
```

Предположим, что программа, которая содержит эти шаблоны, также включает следующий код:

```
struct blot { int a; char b[10]; };
blot ink = { 25, "spots" };
...
recycle(&ink);  // адрес структуры
```

Вызов `recycle(&ink)` соответствует шаблону #1, в котором `Type` интерпретируется как `blot *`. Тот же вызов соответствует и шаблону #2, но на этот раз `Type` будет `ink`. Это сочетание передает два неявных экземпляра, `recycle<blot *>(blot *)` и `recycle<blot>(blot *)`, в пул подходящих функций.

Из этих двух вариантов шаблон `recycle<blot *>(blot *)` является более специализированным, поскольку он предполагает меньшее количество преобразований в процессе генерирования. Другими словами, шаблон #2 уже явно заявил, что аргументом функции является указатель на `Type`, так что `Type` может прямо идентифицироваться как `blot`. Однако шаблон #1 имеет `Type` как аргумент функции, поэтому `Type` должен интерпретироваться как указатель на `blot`. То есть в шаблоне #2 `Type` уже специализирован как указатель, отсюда происходит выражение "более специализированный".

Правила для нахождения наиболее специализированного шаблона называются *правилами частичного упорядочивания* для шаблонов функций. Как и явное создание экземпляров, они являются дополнением языка в стандарте C++98.

Пример использования правил частичного упорядочивания

Рассмотрим завершенную программу, которая использует правила частичного упорядочивания для идентификации применяемого определения шаблона. Листинг 8.14 содержит два определения шаблонов, отображающих содержимое массива. Первое определение (шаблон A) предполагает, что передаваемый в качестве аргумента массив содержит данные, которые следует отобразить. Второе определение (шаблон B) предполагает, что элементы массива являются указателями на отображаемые данные.

Листинг 8.14. `tempover.cpp`

```
// tempover.cpp -- перегрузка шаблонов
#include <iostream>

template <typename T>           // шаблон A
void ShowArray(T arr[], int n);

template <typename T>           // шаблон B
void ShowArray(T * arr[], int n);

struct debts
{
    char name[50];
    double amount;
};
```

```
int main()
{
   using namespace std;
   int things[6] = {13, 31, 103, 301, 310, 130};
   struct debts mr_E[3] =
   {
      {"Ima Wolfe", 2400.0},
      {"Ura Foxe", 1300.0},
      {"Iby Stout", 1800.0}
   };
   double * pd[3];

   // Установка указателей на члены amount структур в mr_E
   for (int i = 0; i < 3; i++)
      pd[i] = &mr_E[i].amount;
   cout << "Listing Mr. E's counts of things:\n";

   // things - массив значений int
   ShowArray(things, 6);      // использует шаблон A
   cout << "Listing Mr. E's debts:\n";

   // pd - массив указателей на double
   ShowArray(pd, 3);          // использует шаблон B (более специализированный)
   return 0;
}
template <typename T>
void ShowArray(T arr[], int n)
{
   using namespace std;
   cout << "template A\n";
   for (int i = 0; i < n; i++)
      cout << arr[i] << ' ';
   cout << endl;
}
template <typename T>
void ShowArray(T * arr[], int n)
{
   using namespace std;
   cout << "template B\n";
   for (int i = 0; i < n; i++)
      cout << *arr[i] << ' ';
   cout << endl;
}
```

Рассмотрим такой вызов функции:

```
ShowArray(things, 6);
```

Идентификатор `things` представляет собой имя массива элементов `int`, поэтому приведенный вызов соответствует следующему шаблону, где `T` получает тип `int`:

```
template <typename T>          // шаблон A
void ShowArray(T arr[], int n);
```

Далее рассмотрим еще один вызов функции:

```
ShowArray(pd, 3);
```

Здесь `pd` — это имя массива элементов `double *`. Этот вызов соответствует шаблону A:

```
template <typename T>          // шаблон A
void ShowArray(T arr[], int n);
```

Вместо T подставляется тип double *. В этом случае шаблонная функция отобразит содержимое массива pd, т.е. три адреса. Приведенный вызов функции также может быть сопоставлен с шаблоном B:

```
template <typename T>          // шаблон B
void ShowArray(T * arr[], int n);
```

Здесь T получает тип double, а функция отображает разыменованные элементы *arr[i] — значения типа double, на которые указывают элементы массива. Из двух шаблонов более специализированным является шаблон B, поскольку он построен исходя из предположения, что массив содержит указатели. Поэтому именно шаблон B и будет использоваться. Ниже показан вывод программы из листинга 8.14:

```
Listing Mr. E's counts of things:
template A
13 31 103 301 310 130
Listing Mr. E's debts:
template B
2400 1300 1800
```

Если удалить из программы шаблон B, компилятор будет использовать шаблон A для вывода содержимого массива pd, поэтому список будет содержать адреса, а не значения. Попробуйте сами и посмотрите, что получится.

Кратко подведем итоги. Процесс разрешения перегрузки ищет функцию, которая будет наилучшим соответствием. Если существует лишь одна такая функция, она и выбирается. Если вариантов несколько, но только одна функция является нешаблонной, она и выбирается. Когда кандидатов несколько, и все они являются шаблонными функциями, выбирается наиболее специализированная из них. Если существуют две или больше в одинаковой степени соответствующих нешаблонных функции, либо две или больше одинаково подходящих шаблонных функции с одной и той же степенью специализации, вызов функции рассматривается как неоднозначный и приводит к ошибке. При отсутствии функций, соответствующих вызову, также возникает ошибка.

Обеспечение необходимого выбора

В некоторых обстоятельствах можно заставить компилятор сделать необходимый вам выбор, правильно написав вызов функции. Взгляните на листинг 8.15, в котором, кстати, устранен прототип шаблона, а определение шаблонной функции помещено в начало файла. Как и в случае обычных функций, определение шаблонной функции может действовать в качестве своего прототипа, если оно находится перед использованием функции.

Листинг 8.15. choices.cpp

```
// choices.cpp — выбор шаблона
#include <iostream>
template<class T>              // или template <typename T>
T lesser(T a, T b)             // #1
{
   return a < b ? a : b;
}
int lesser (int a, int b)      // #2
{
   a = a < 0 ? -a : a;
   b = b < 0 ? -b : b;
   return a < b ? a : b;
}
```

```
int main()
{
   using namespace std;
   int m = 20;
   int n = -30;
   double x = 15.5;
   double y = 25.9;
   cout << lesser(m, n) << endl;          // используется #2
   cout << lesser(x, y) << endl;          // используется #1 с double
   cout << lesser<>(m, n) << endl;        // используется #1 с int
   cout << lesser<int>(x, y) << endl;     // используется #1 с int
   return 0;
}
```

(В последнем вызове функции выполняется преобразование `double` в `int`, и некоторые компиляторы выдают предупреждение об этом.)

Ниже показан вывод программы из листинга 8.15:

```
20
15.5
-30
15
```

В листинге 8.15 предоставлен шаблон, который возвращает меньшее из двух значений, и стандартная функция, возвращающая меньший модуль из двух значений. Если определение функции находится перед ее первым использованием, оно действует как прототип, так что в этом примере прототипы опущены. Рассмотрим следующий оператор:

```
cout << lesser(m, n) << endl;          // используется #2
```

Аргументы в этом вызове соответствуют как шаблонной функции, так и нешаблонной функции, поэтому выбирается нешаблонная функция, которая возвращает значение `20`.

Следующий вызов функции соответствует шаблону, при этом `T` становится `double`:

```
cout << lesser(x, y) << endl;          // используется #1 с double
```

Теперь взгляните на такой оператор:

```
cout << lesser<>(m, n) << endl;        // используется #1 с int
```

Наличие угловых скобок в `lesser<>(m, n)` указывает, что компилятор должен выбрать шаблонную функцию вместо нешаблонной, и компилятор, отметив, что фактические аргументы имеют тип `int`, создает экземпляр шаблона с использованием `int` для `T`.

Наконец, рассмотрим следующий оператор:

```
cout << lesser<int>(x, y) << endl;  // используется #1 с int
```

Здесь мы имеем запрос на явное создание экземпляра с применением `int` для `T`, и эта функция будет использоваться. Значения `x` и `y` приводятся к типу `int`, и функция возвращает значение `int`, из-за чего программа отображает `15` вместо `15.5`.

Функции с множеством аргументов-типов

Когда вызов функции с множеством аргументов сопоставляется с прототипами, содержащими несколько аргументов-типов, ситуация значительно усложняется. Компилятору приходится проверять соответствия всех аргументов. Если удается най-

ти функцию, которая подходит лучше других кандидатов, она и будет выбрана. Одна функция имеет приоритет перед другой, если хотя бы один ее аргумент имеет приоритет перед аргументом другой функции, а все остальные аргументы обладают, по меньшей мере, одинаковыми приоритетами.

Темой настоящей книги не является исследование сложных примеров поиска наилучшего соответствия. Рассмотренные выше правила охватывают все возможные сочетания прототипов и шаблонов функций.

Эволюция шаблонных функций

В начале становления C++ большинство людей не предвидели, насколько мощными и полезными окажутся шаблонные функции и шаблонные классы. (Возможно, они не могли это представить даже в своем воображении.) Однако умелые и преданные делу программисты отбросили ограничения технологий шаблонов и расширили идеи того, что с их помощью можно делать. Отзывы от тех, кто постоянно работал с шаблонами, привели к изменениям, которые были включены в стандарт C++98, а также к ряду добавлений в стандартную библиотеку шаблонов (STL). С тех пор программисты, использующие шаблоны, продолжили изучать предлагаемые ими возможности, и периодически сталкивались с ограничениями. Благодаря обратной связи с ними, в стандарт C++11 были внесены некоторые изменения. Ниже мы рассмотрим несколько таких проблем вместе с их решениями.

Каким должен быть тип?

Одна из проблем связана с тем, что при написании шаблонной функции в C++98 не всегда возможно знать, какой тип использовать в объявлении. Взгляните на следующий частичный пример:

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
    ...
    ?тип? xpy = x + y;
    ...
}
```

Каким должен быть тип для `xpy`? Мы не знаем заранее, как может использоваться `ft()`. Подходящим типом может быть `T1`, `T2` или какой-то совершенно другой тип. Например, `T1` может быть `double`, а `T2` — `int`, тогда типом их суммы будет `double`. Или же `T1` может быть `short`, а `T2` — `int`, и в этом случае типом суммы окажется `int`. Предположим, что `T1` является `short`, а `T2` — `char`. Тогда сложение приводит к автоматическому целочисленному расширению и результирующим типом будет `int`. Кроме того, операция `+` может быть перегружена для структур и классов, дополнительно усложняя варианты выбора. Таким образом, в C++98 для типа `xpy` очевидный выбор отсутствует.

Ключевое слово `decltype` *(C++11)*

В стандарте C++11 проблема решается с помощью нового ключевого слова `decltype`. Оно может использоваться следующим образом:

```
int x;
decltype(x) y;  // делает тип y тем же, что и y x
```

Аргументом `decltype` может быть выражение, поэтому в примере с `ft()` можно написать такой код:

```
decltype(x + y) xpy;       // делает тип xpy тем же, что и у x + y
xpy = x + y;
```

В качестве альтернативы эти два оператора могут быть скомбинированы внутри инициализации:

```
decltype(x + y) xpy = x + y;
```

Таким образом, шаблон `ft()` можно скорректировать, как показано ниже:

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
   ...
   decltype(x + y) xpy = x + y;
   ...
}
```

Средство `decltype` несколько сложнее, чем может показаться на основе приведенных выше примеров. Для выбора типа компилятор должен пройти контрольный список. Предположим, что имеется такой код:

```
decltype(выражение) var;
```

Ниже представлена слегка упрощенная версия этого списка.

Фаза 1. Если *выражение* является идентификатором без дополнительных круглых скобок, тогда `var` получит тот же самый тип, что у идентификатора, включая его квалификаторы, такие как `const`:

```
double x = 5.5;
double y = 7.9;
double &rx = x;
const double * pd;
decltype(x) w;             // w имеет тип double
decltype(rx) u = y;        // u имеет тип double &
decltype(pd) v;            // v имеет тип const double *
```

Фаза 2. Если *выражение* является вызовом функции, тогда `var` имеет тип возвращаемого значения этой функции:

```
long indeed(int);
decltype (indeed(3)) m;   // m имеет тип int
```

На заметку!

Выражение в форме вызова функции не вычисляется. В этом случае компилятор берет возвращаемый тип из прототипа функции; в действительном вызове функции необходимости нет.

Фаза 3. Если *выражение* является lvalue, тогда `var` будет ссылкой на тип выражения. Может показаться, что в предыдущих примерах переменная `w` должна была иметь ссылочный тип, учитывая, что `w` является lvalue. Однако вспомните, что этот случай уже был перехвачен на фазе 1. На данной фазе *выражение* не может быть идентификатором без дополнительных круглых скобок. А чем же тогда? Одна из очевидных возможностей — идентификатор с дополнительными круглыми скобками:

```
double xx = 4.4;
decltype ((xx)) r2 = xx;  // r2 имеет тип double &
decltype(xx) w = xx;      // w имеет тип double (соответствие на фазе 1)
```

Кстати, круглые скобки не изменяют обычное значение или значение lvalue выражения. Например, следующие два оператора дают один и тот же эффект:

```
xx = 98.6;
(xx) = 98.6;                    // () не влияют на использование xx
```

Фаза 4. Если ни один из предыдущих специальных случаев не применим, тогда `var` имеет тот же тип, что и *выражение*:

```
int j = 3;
int &k = j
int &n = j;
decltype(j+6) i1;               // i1 имеет тип int
decltype(100L) i2;              // i2 имеет тип long
decltype(k+n) i3;               // i3 имеет тип int;
```

Обратите внимание, что хотя `k` и `n` являются ссылками, выражение `k+n` — не ссылка; это просто сумма двух значений `int`, т.е. `int`.

Когда необходимо более одного объявления, можно воспользоваться `typedef` с `decltype`:

```
template<class T1, class T2>
void ft(T1 x, T2 y)
{
   ...
   typedef decltype(x + y) xytype;
   xytype xpy = x + y;
   xytype arr[10];
   xytype & rxy = arr[2];       // rxy - ссылка
   ...
}
```

Альтернативный синтаксис для функций (хвостовой возвращаемый тип C++11)

Механизм `decltype` сам по себе оставляет нерешенной другую связанную проблему. Рассмотрим следующую незавершенную шаблонную функцию:

```
template<class T1, class T2>
?тип? gt(T1 x, T2 y)
{
   ...
   return x + y;
}
```

Мы снова не знаем заранее тип результата сложения `x` и `y`. Может показаться, что для возвращаемого типа подойдет `decltype(x + y)`. К сожалению, в этой точке кода параметры `x` и `y` еще не определены, поэтому они находятся за пределами контекста (не являются видимыми и не доступны для использования компилятором). Спецификатор `decltype` должен следовать *после* того, как параметры объявлены. Чтобы сделать это возможным, в C++11 вводится новый синтаксис для объявления и определения функций. Ниже показано его применение на примере встроенных типов. Прототип

```
double h(int x, float y);
```

может быть записан с помощью альтернативного синтаксиса следующим образом:

```
auto h(int x, float y) -> double;
```

Как видите, возвращаемый тип перемещен за объявления параметров. Комбинация `-> double` называется *хвостовым возвращаемым типом* (trailing return type). Ключевое слово `auto` здесь выступает в новой роли, введенной в C++11, и является заполнителем для типа, предоставляемого хвостовым возвращаемым типом. Та же форма будет использоваться в определении функции:

```
auto h(int x, float y) -> double
{/* тело функции */};
```

Комбинируя новый синтаксис с `decltype`, указать возвращаемый тип для функции `gt()` можно так:

```
template<class T1, class T2>
auto gt(T1 x, T2 y) -> decltype(x + y)
{
    ...
    return x + y;
}
```

Теперь `decltype` находится после объявлений параметров, поэтому `x` и `y` являются видимыми и доступными для использования.

Резюме

Язык C++ расширил возможности функций C. Ключевое слово `inline` в определении функции и размещение этого определения до первого вызова функции указывает компилятору C++ обращаться с данной функцией как со встроенной. Иначе говоря, вместо перехода к отдельному разделу кода для выполнения функции, компилятор встраивает взамен каждого вызова функции соответствующий код. Этот механизм встраивания должен использоваться только в тех случаях, когда код функции достаточно краткий.

Ссылочная переменная — это разновидность скрытого указателя, который позволяет создавать псевдоним (второе имя) для переменной. Ссылочные переменные главным образом используются в качестве аргументов функций, которые обрабатывают структуры и объекты классов. Обычно идентификатор, объявленный как ссылка на определенный тип, может указывать только на данные этого типа. Однако когда один класс является производным от другого (например, класс `ofstream` унаследован от `ostream`), ссылка на базовый тип может также указывать на производный тип.

Прототипы C++ позволяют определять значения по умолчанию для аргументов. Если в вызове функции опущен соответствующий аргумент, программа использует его значение по умолчанию. Если в обращении к функции значение аргумента указано, программа использует его вместо значения по умолчанию. Аргументы по умолчанию могут предоставляться в списке аргументов только справа налево. Таким образом, если вы указываете значение по умолчанию для определенного аргумента, то при этом должны быть указаны значения по умолчанию для всех аргументов, расположенных справа от него.

Сигнатурой функции является ее список аргументов. Можно определить две функции с одним и тем же именем при условии, что они имеют разные сигнатуры. Это называется *полиморфизмом* или *перегрузкой функций*. Как правило, перегрузка функции осуществляется для того, чтобы обеспечить единообразную обработку различных типов данных.

Шаблоны функций автоматизируют процесс перегрузки функций. Функция определяется с применением обобщенного типа данных и отдельного алгоритма, а ком-

пилятор генерирует соответствующие определения функций для конкретных типов аргументов, которые используются в программе.

Вопросы для самоконтроля

1. Какие разновидности функций являются хорошими кандидатами на то, чтобы быть встроенными?
2. Предположим, что функция `song()` имеет следующий прототип:

   ```
   void song(char * name, int times);
   ```

   а. Как модифицировать этот прототип, чтобы для переменной `times` по умолчанию принималось значение 1?

   б. Какие изменения следует внести в определение функции?

   в. Можно ли переменной `name` присвоить используемое по умолчанию значение `"O, My Papa"`?
3. Напишите перегруженные версии функции `iquote()`, которая отображает аргументы, заключенные в двойные кавычки. Напишите три версии: одну для аргумента типа `int`, другую для аргумента типа `double` и третью для аргумента типа `string`.
4. Пусть имеется следующая структура:

   ```
   struct box
   {
      char maker[40];
      float height;
      float width;
      float length;
      float volume;
   };
   ```

   а. Напишите функцию, которая имеет формальный аргумент — ссылку на структуру `box` и отображает значение каждого члена структуры.

   б. Напишите функцию, которая имеет формальный аргумент — ссылку на структуру `box` и устанавливает член `volume` в результат произведения членов `height`, `width` и `length`.
5. Какие изменения понадобится внести в листинг 7.15, чтобы функции `fill()` и `show()` использовали ссылочные параметры?
6. Ниже дано описание результатов, которые требуется обеспечить. Укажите, может ли каждый из них быть получен с помощью аргументов по умолчанию, путем перегрузки функций, тем и другим способом, или же можно обойтись без этих средств. Предоставьте необходимые прототипы.

   а. Функция `mass(density, volume)` возвращает массу тела, имеющего плотность `density` и объем `volume`, а функция `mass(density)` возвращает массу тела, имеющего плотность `density` и объем `1.0` кубический метр. Все величины имеют тип `double`.

   б. Вызов `repeat(10, "I'm OK")` отображает указанную строку 10 раз, а вызов `repeat("But you're kind of stupid")` отображает заданную строку 5 раз.

в. Вызов `average(3,6)` возвращает среднее значение типа `int` двух аргументов `int`, а вызов `average(3.0, 6.0)` — среднее значение типа `double` двух значений `double`.

г. Вызов `mangle("I'm glad to meet you")` возвращает символ `I` или указатель на строку `"I'm glad to meet you"` в зависимости от того, присваивается возвращаемое значение переменной типа `char` или переменной типа `char*`.

7. Напишите шаблон функции, которая возвращает больший из двух ее аргументов.
8. Используя шаблон из вопроса 7 и структуру `box` из вопроса 4, предоставьте специализацию шаблона, которая принимает два аргумента типа `box` и возвращает тот из них, у которого больше значение `volume`.
9. Какие типы назначены переменным `v1`, `v2`, `v3`, `v4` и `v5` в следующем коде (предполагается, что код является частью завершенной программы)?

```
int g(int x);
...
float m = 5.5f;
float & rm = m;
decltype(m) v1 = m;
decltype(rm) v2 = m;
decltype((m)) v3 = m;
decltype (g(100)) v4;
decltype (2.0 * m) v5;
```

Упражнения по программированию

1. Напишите функцию, которая обычно принимает один аргумент — адрес строки и выводит эту строку один раз. Однако если задан второй аргумент типа `int`, не равный нулю, то эта функция выводит строку столько раз, сколько было осуществлено вызовов этой функции к моменту ее данного вызова. (Обратите внимание, что количество выводимых строк не равно значению второго аргумента, оно равно числу вызовов функции к моменту последнего вызова.) Действительно, это не слишком полезная функция, но она заставит применить некоторые из методов, рассмотренных в данной главе. Напишите простую программу для демонстрации этой функции.
2. Структура `CandyBar` содержит три члена. Первый член хранит название коробки конфет. Второй — ее вес (который может иметь дробную часть), а третий — количество калорий (целое значение). Напишите программу, использующую функцию, которая принимает в качестве аргументов ссылку на `CandyBar`, указатель на `char`, значение `double` и значение `int`. Функция использует три последних значения для установки соответствующих членов структуры. Три последних аргумента должны иметь значения по умолчанию: `"Millennium Munch"`, `2.85` и `350`. Кроме того, программа должна использовать функцию, которая принимает в качестве аргумента ссылку на `CandyBar` и отображает содержимое этой структуры. Где необходимо, используйте `const`.
3. Напишите функцию, которая принимает ссылку на объект `string` в качестве параметра и преобразует содержимое `string` в символы верхнего регистра. Используйте функцию `toupper()`, описанную в табл. 6.4 (см. главу 6). Напишите программу, использующую цикл, которая позволяет проверить работу функции для разного ввода. Пример вывода может выглядеть следующим образом:

```
Enter a string (q to quit): go away
GO AWAY
Next string (q to quit): good grief!
GOOD GRIEF!
Next string (q to quit): q
Bye.
```

4. Ниже представлена общая структура программы:

```
#include <iostream>
using namespace std;
#include <cstring>       // для strlen(), strcpy()

struct stringy {
   char * str;           // указывает на строку
   int ct;               // длина строки (не считая символа '\0')
};

// Здесь размещаются прототипы функций set() и show()

int main()
{
   stringy beany;
   char testing[] = "Reality isn't what it used to be.";
   set(beany, testing); // первым аргументом является ссылка,
   // Выделяет пространство для хранения копии testing,
   // использует элемент типа str структуры beany как указатель
   // на новый блок, копирует testing в новый блок и
   // создает элемент ct структуры beany
   show(beany);         // выводит строковый член структуры один раз
   show(beany, 2);      // выводит строковый член структуры два раза
   testing[0] = 'D';
   testing[1] = 'u';
   show(testing);       // выводит сроку testing один раз
   show(testing, 3);    // выводит строку testing три раза
   show("Done!");
   return 0;
```

Завершите программу, создав соответствующие функции и прототипы. Обратите внимание, что в программе должны быть две функции `show()`, и каждая из них использует аргументы по умолчанию. Где необходимо, используйте `const`. Функция `set()` должна использовать операцию `new` для выделения достаточного пространства памяти под хранение заданной строки. Используемые здесь методы аналогичны методам, применяемым при проектировании и реализации классов. (В зависимости от используемого компилятора, может понадобиться изменить имена заголовочных файлов и удалить директиву `using`.)

5. Напишите шаблонную функцию `max5()`, которая принимает в качестве аргумента массив из пяти элементов типа `T` и возвращает наибольший элемент в массиве. (Поскольку размер массива фиксирован, его можно жестко закодировать в цикле, а не передавать в виде аргумента.) Протестируйте функцию в программе с использованием массива из пяти значений `int` и массива из пяти значений `double`.

6. Напишите шаблонную функцию `maxn()`, которая принимает в качестве аргумента массив элементов типа `T` и целое число, представляющее количество элементов в массиве, а возвращает элемент с наибольшим значением. Протестируйте ее работу в программе, которая использует этот шаблон с массивом из шести значений `int` и массивом из четырех значений `double`. Программа также должна включать специализацию, которая использует массив указателей на `char` в качестве первого аргумента и количество указателей — в качестве второго, а затем возвращает адрес самой длинной строки. Если имеется более одной строки наибольшей длины, функция должна вернуть адрес первой из них. Протестируйте специализацию на массиве из пяти указателей на строки.
7. Измените программу из листинга 8.14 так, чтобы использовать две шаблонных функции по имени `SumArray()`, возвращающие сумму содержимого массива вместо его отображения. Программа должна сообщать общее количество предметов и сумму всех задолженностей (`debts`).

9

Модели памяти и пространства имен

В ЭТОЙ ГЛАВЕ...

- Раздельная компиляция программ
- Продолжительность хранения, область видимости и компоновка
- Операция `new` с размещением
- Пространства имен

Язык C++ предлагает множество способов хранения данных в памяти. Имеется возможность выбора длительности хранения данных в памяти (продолжительность существования области хранения) и определения частей программы, имеющих доступ к данным (область видимости и связывание). Операция `new` позволяет динамически выделять память, а операция `new` с размещением является ее вариацией. Возможности пространства имен C++ обеспечивают дополнительный контроль над доступом к данным. Крупные программы обычно состоят из нескольких файлов исходного кода, которые могут совместно использовать определенные данные. Поскольку в таких программах применяется раздельная компиляция файлов, эта глава начинается с освещения данной темы.

Раздельная компиляция

Язык C++, как и C, позволяет и даже поощряет размещение функций программы в отдельных файлах. Как говорилось в главе 1, файлы можно компилировать раздельно, а затем связывать их с конечным продуктом — исполняемой программой. (Как правило, компилятор C++ не только компилирует программы, но и управляет работой компоновщика.) При изменении только одного файла можно перекомпилировать лишь этот файл и затем связать его с ранее скомпилированными версиями других файлов. Этот механизм облегчает работу с крупными программами. Более того, большинство сред программирования на C++ предоставляют дополнительные средства, упрощающие такое управление. Например, в системах Unix и Linux имеется программа `make`, хранящая сведения обо всех файлах, от которых зависит программа, и о времени их последней модификации. После запуска `make` обнаруживает изменения в исходных файлах с момента последней компиляции, а затем предлагает выполнить соответствующие действия, необходимые для воссоздания программы. Большинство интегрированных сред разработки (integrated development environment — IDE), включая Embarcadero C++ Builder, Microsoft Visual C++, Apple Xcode и Freescale CodeWarrior, предоставляют аналогичные средства, доступ к которым осуществляется с помощью меню Project (Проект).

Рассмотрим простой пример. Вместо того чтобы разбирать детали компиляции, которые зависят от реализации, давайте сосредоточим внимание на более общих аспектах, таких как проектирование.

Предположим, что решено разделить программу из листинга 7.12 на части и поместить используемые ею функции в отдельный файл. Напомним, что эта программа преобразует прямоугольные координаты в полярные, после чего отображает результат. Нельзя просто вырезать из исходного файла часть кода после окончания функции `main()`. Дело в том, что `main()` и другие две функции используют одни и те же объявления структур, поэтому необходимо поместить эти объявления в оба файла. При простом наборе объявлений в коде можно допустить ошибку. Но даже если объявления скопированы безошибочно, при последующих модификациях нужно будет не забыть внести изменения в оба файла. Одним словом, разделение программы на несколько файлов создает новые проблемы.

Кому нужны дополнительные сложности? Только не разработчикам C и C++. Для решения подобных проблем была предоставлена директива `#include`. Вместо того чтобы помещать объявления структур в каждый файл, их можно разместить в заголовочном файле, а затем включать его в каждый файл исходного кода. Таким образом, изменения в объявление структуры будут вноситься только один раз в заголовочный файл. Кроме того, в заголовочный файл можно помещать прототипы функций.

Итак, исходную программу можно разбить на три части:

- заголовочный файл, содержащий объявления структур и прототипы функций, которые используют эти структуры;
- файл исходного кода, содержащий код функций, которые работают со структурами;
- файл исходного кода, содержащий код, который вызывает функции работы со структурами.

Такая стратегия может успешно применяться для организации программы. Если, например, создается другая программа, которая пользуется теми же самыми функциями, достаточно включить в нее заголовочный файл и добавить файл с функциями в проект или список make. К тому же такая организация программы соответствует принципам объектно-ориентированного программирования (ООП). Первый файл – заголовочный – содержит определения пользовательских типов. Второй файл содержит код функций для манипулирования типами, определенными пользователем. Вместе оба файла формируют пакет, который можно использовать в различных программах.

В заголовочный файл не следует помещать определения функций или объявления переменных. Хотя в простейших проектах такой подход может работать, обычно он приводит к проблемам. Например, если в заголовочном файле содержится определение функции, и этот заголовочный файл включен в два других файла, которые являются частью одной программы, в этой программе окажется два определения одной и той же функции, что вызовет ошибку, если только функция не является встроенной. В заголовочных файлах обычно содержится следующее:

- прототипы функций;
- символические константы, определенные с использованием `#define` или `const`;
- объявления структур;
- объявления классов;
- объявления шаблонов;
- встроенные функции.

Объявления структур можно помещать в заголовочные файлы, поскольку они не создают переменные, а только указывают компилятору, как создавать структурную переменную, когда она объявляется в файле исходного кода. Подобно этому объявления шаблонов – это не код, который нужно компилировать, а инструкции для компилятора, указывающие, каким образом генерировать определения функций, чтобы они соответствовали вызовам функций, встречающимся в исходном коде. Данные, объявленные как `const`, и встроенные функции имеют специальные свойства связывания (вскоре они будут рассмотрены), которые позволяют размещать их в заголовочных файлах, не вызывая при этом каких-либо проблем.

В листингах 9.1, 9.2 и 9.3 показан результат разделения программы из листинга 7.12 на отдельные части. Обратите внимание, что при включении заголовочного файла используется запись `"coordin.h"`, а не `<coordin.h>`. Если имя файла помещено в угловые скобки, компилятор C++ ищет его в той части базовой файловой системы, где расположены стандартные заголовочные файлы. Но когда имя файла представлено в двойных кавычках, компилятор сначала ищет файл в текущем рабочем каталоге или в каталоге с исходным кодом (либо в другом аналогичном месте, которое зависит от версии компилятора). Не обнаружив заголовочный файл там, он ищет его в стандартном местоположении. Таким образом, при включении собственных заголовочных файлов должны использоваться двойные кавычки, а не угловые скобки.

На рис. 9.1 показаны шаги по сборке этой программы в системе Unix. Обратите внимание, что пользователь только выдает команду компиляции `CC`, а остальные действия выполняются автоматически. Компиляторы командной строки g++, gpp и Borland C++ (`bcc32.exe`) ведут себя аналогичным образом. Среды разработки Apple Xcode, Embarcadero C++ Builder и Microsoft Visual C++ в сущности выполняют те же самые действия, однако, как упоминалось в главе 1, процесс инициируется по-другому, с помощью команд меню, которые позволяют создавать проект и ассоциировать с ним файлы исходного кода. Обратите внимание, что в проекты добавляются только файлы исходного кода, но не заголовочные файлы. Дело в том, что заголовочными файлами управляет директива `#include`. Кроме того, не следует использовать директиву `#include` для включения файлов исходного кода, поскольку это может привести к дублированным объявлениям.

Внимание!

В интегрированных средах разработки не добавляйте заголовочные файлы в список проекта и не используйте директиву `#include` для включения одних файлов исходного кода в другие файлы исходного кода.

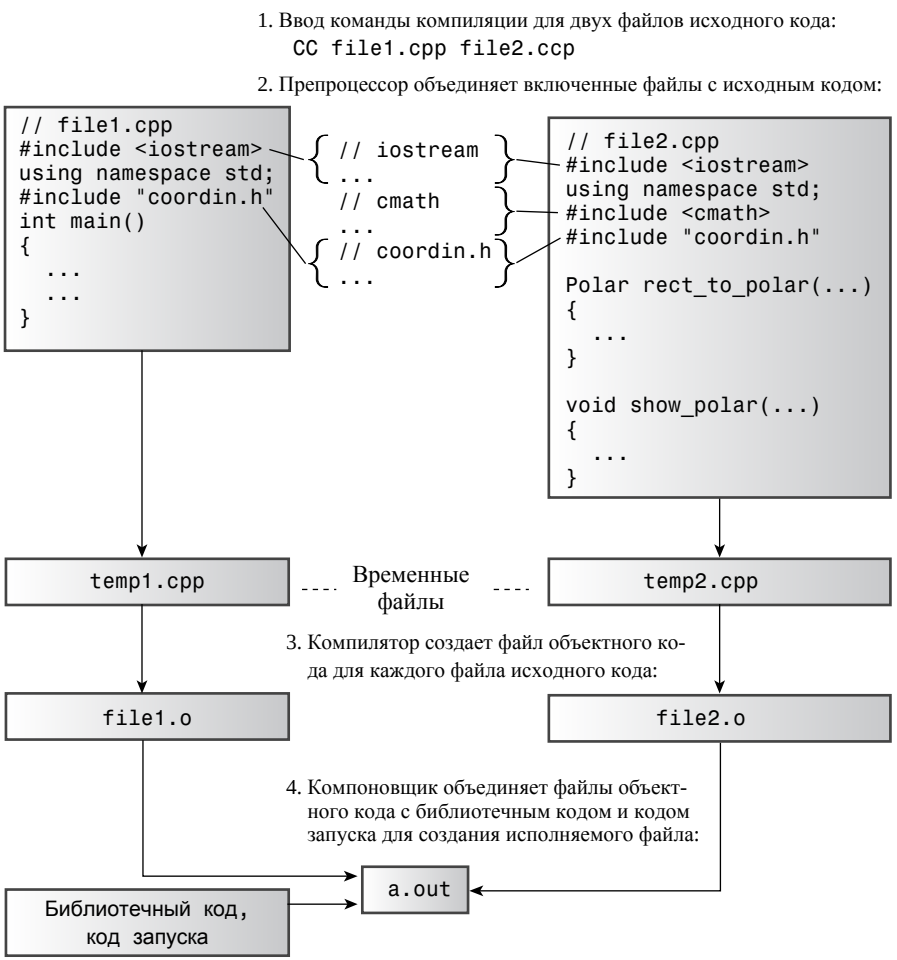


***Рис. 9.1.** Компиляция многофайловых программ C++ в системе Unix*

Листинг 9.1. coordin.h

```
// coordin.h -- шаблоны структур и прототипы функций
// шаблоны структур
#ifndef COORDIN H
#define COORDIN H

struct polar
{
   double distance;      // расстояние от исходной точки
   double angle;         // направление от исходной точки
};
struct rect
{
   double x;             // расстояние по горизонтали от исходной точки
   double y;             // расстояние по вертикали от исходной точки
};

// прототипы
polar rect_to_polar(rect xypos);
void show_polar(polar dapos);

#endif
```

Управление заголовочными файлами

Заголовочный файл должен включаться в файл только один раз. Это кажется простым требованием, которое легко запомнить и придерживаться, тем не менее, можно непреднамеренно включить заголовочный файл несколько раз, даже не подозревая об этом. Например, предположим, что используется заголовочный файл, который включает другой заголовочный файл. В C/C++ существует стандартный прием, позволяющий избежать многократных включений заголовочных файлов. Он основан на использовании директивы препроцессора `#ifndef` (*if not defined* — если не определено). Показанный ниже фрагмент кода обеспечивает обработку операторов, находящихся между директивами `#ifndef` и `#endif`, только в случае, если имя `COORDIN_H_` не было определено ранее с помощью директивы препроцессора `#define`:

```
#ifndef COORDIN_H_
...
#endif
```

Обычно директива `#define` используется для создания символических констант, как в следующем примере:

```
#define MAXIMUM 4096
```

Однако для определения имени достаточно просто указать директиву `#define` с этим именем:

```
#define COORDIN_H_
```

Прием, применяемый в листинге 9.1, предусматривает помещение содержимого файла внутрь `#ifndef`:

```
#ifndef COORDIN_H_
#define COORDIN_H_
// здесь размещается содержимое включаемого файла
#endif
```

Когда компилятор впервые сталкивается с этим файлом, имя `COORDIN_H_` должно быть неопределенным. (Во избежание совпадения с существующими именами, имя строится на основе имени включаемого файла и нескольких символов подчеркивания.)

В этом случае компилятор будет обрабатывать код между директивами `#ifndef` и `#endif`, что, собственно, и требуется. Во время обработки компилятор читает строку с директивой, определяющей имя `COORDIN_H_`. Если затем компилятор обнаруживает второе включение `coordin.h` в том же самом файле, он замечает, что имя `COORDIN_H_` уже определено, и переходит к строке, следующей после `#endif`. Обратите внимание, что данный прием не предотвращает повторного включения файла. Вместо этого он заставляет компилятор игнорировать содержимое всех включений кроме первого. Такая методика защиты используется в большинстве стандартных заголовочных файлов C и C++. Если ее не применять, одна и та же структура, например, окажется объявленной в файле дважды, что приведет к ошибке компиляции.

Листинг 9.2. `file1.cpp`

```
// file1.cpp -- пример программы, состоящей из трех файлов
#include <iostream>
#include "coordin.h" // шаблоны структур, прототипы функций
using namespace std;
int main()
{
   rect rplace;
   polar pplace;
   cout << "Enter the x and y values: ";           // ввод значений x и y
   while (cin >> rplace.x >> rplace.y)             // ловкое использование cin
   {
      pplace = rect_to_polar(rplace);
      show_polar(pplace);
      cout << "Next two numbers (q to quit): ";
            // ввод следующих двух чисел (q для завершения)
   }
   cout << "Done.\n";
   return 0;
}
```

Листинг 9.3. `file2.cpp`

```
// file2.cpp -- содержит функции, вызываемые в file1.cpp
#include <iostream>
#include <cmath>
#include "coordin.h"     // шаблоны структур, прототипы функций

// Преобразование прямоугольных координат в полярные
polar rect_to_polar(rect xypos)
{
   using namespace std;
   polar answer;
   answer.distance =
      sqrt( xypos.x * xypos.x + xypos.y * xypos.y);
   answer.angle = atan2(xypos.y, xypos.x);
   return answer;         // возврат структуры polar
}

// Отображение полярных координат с преобразованием радиан в градусы
void show_polar (polar dapos)
{
   using namespace std;
   const double Rad_to_deg = 57.29577951;
   cout << "distance = " << dapos.distance;
   cout << ", angle = " << dapos.angle * Rad_to_deg;
   cout << " degrees\n";
}
```

В результате компиляции и компоновки этих двух файлов исходного кода и нового заголовочного файла получается исполняемая программа. Ниже приведен пример ее выполнения:

```
Enter the x and y values: 120 80
distance = 144.222, angle = 33.6901 degrees
Next two numbers (q to quit): 120 50
distance = 130, angle = 22.6199 degrees
Next two numbers (q to quit): q
```

Кстати, хотя мы обсудили раздельную компиляцию в терминах файлов, в стандарте C++ вместо термина *файл* используется термин *единица трансляции*, чтобы сохранить более высокую степень обобщенности; файловая модель — это не единственный способ организации информации в компьютере. Для простоты в этой книге будет применяться термин "файл", но помните, что под этим понимается также и "единица трансляции".

Связывание с множеством библиотек

Стандарт C++ предоставляет каждому разработчику компилятора возможность самостоятельной реализации декорирования имен (см. врезку "Что такое декорирование имен?" в главе 8), поэтому следует учитывать, что связывание двоичных модулей (файлов объектного кода), созданных различными компиляторами, скорее всего, не будет успешным. Другими словами, для одной и той же функции два компилятора сгенерируют различные декорированные имена. Такое различие в именах не позволит компоновщику найти соответствие между вызовом функции, сгенерированной одним компилятором, и определением функции, сгенерированным другим компилятором. Перед компоновкой скомпилированных модулей нужно обеспечить, чтобы каждый объектный файл или библиотека была сгенерирована одним и тем же компилятором. При наличии исходного кода проблемы компоновки обычно легко решаются за счет повторной компиляции.

Продолжительность хранения, область видимости и компоновка

После обзора многофайловых программ пришло время продолжить рассмотрение моделей памяти, начатое в главе 4. Дело в том, что категории хранения влияют на то, как информация может совместно использоваться разными файлами. Вспомните, что в главе 4 говорилось о памяти. В языке C++ применяются три различных схемы хранения данных (в C++11 их четыре). Эти схемы отличаются между собой продолжительностью нахождения данных в памяти.

- **Автоматическая продолжительность хранения.** Переменные, объявленные внутри определения функции — включая параметры функции — имеют автоматическую продолжительность хранения. Они создаются, когда выполнение программы входит в функцию или блок, где эти переменные определены. После выхода из блока или функции используемая переменными память освобождается. В C++ существуют два вида автоматических переменных.
- **Статическая продолжительность хранения.** Переменные, объявленные за пределами определения функции либо с использованием ключевого слова `static`, имеют статическую продолжительность хранения. Они существуют в течение всего времени выполнения программы. В языке C++ существуют три вида переменных со статической продолжительностью хранения.

- **Потоковая продолжительность хранения (C++11).** В наши дни многоядерные процессоры распространены практически повсеместно. Такие процессоры способны поддерживать множество выполняющихся задач одновременно. Это позволяет программе разделить вычисления на отдельные *потоки*, которые могут быть обработаны параллельно. Переменные, объявленные с ключевым словом `thread_local`, хранятся на протяжении времени существования содержащего их потока. Вопросы параллельного программирования в этой книге не рассматриваются.
- **Динамическая продолжительность хранения**. Память, выделяемая операцией `new`, сохраняется до тех пор, пока она не будет освобождена с помощью операции `delete` или до завершения программы, смотря какое из событий наступит раньше. Эта память имеет динамическую продолжительность хранения и часто называется *свободным хранилищем* или *кучей*.

Далее мы продолжим изучение понятий области видимости переменных (их доступности для программы) и компоновки, которая определяет, какая информация совместно используется разными файлами.

Область видимости и связывание

Область видимости (или *контекст*) определяет доступность имени в пределах файла (единицы трансляции). Например, переменная, определенная в функции, может быть использована только в этой функции, но не в какой-либо другой, в то время как переменная, определенная в файле до определений функций, может применяться во всех функциях. *Связывание* описывает, как имя может разделяться различными единицами трансляции. Имя с *внешним связыванием* может совместно использоваться разными файлами, а имя с *внутренним связыванием* – функциями внутри одного файла. Имена автоматических переменных не имеют никакого связывания, поскольку они не являются разделяемыми.

Переменная C++ может иметь одну из нескольких возможных областей видимости. Переменная с *локальной областью видимости* (которая также называется *областью видимости блока*) известна только внутри блока, где она определена. Вспомните, что блок – это последовательность операторов, заключенная в фигурные скобки. Например, тело функции является блоком, однако в него могут быть вложены и другие блоки. Переменная, имеющая *глобальную область видимости* (которая часто называется *областью видимости файла*), известна во всем файле, начиная с точки, где она определена. Автоматические переменные имеют локальную область видимости, а статические переменные могут иметь различную область видимости в зависимости от того, как они определены. Имена, используемые в *области видимости прототипа функции*, доступны только в пределах круглых скобок, которые содержат список аргументов. (Вот почему не важно, что они собой представляют и присутствуют ли вообще.) Элементы, объявленные в классе, имеют *область видимости класса* (см. главу 10). Переменные, объявленные в пространстве имен, имеют *область видимости пространства имен*. (Теперь, когда в C++ были добавлены пространства имен, глобальная область видимости стала частным случаем области видимости пространства имен.)

Функции C++ могут иметь область видимости класса или область видимости пространства имен, включая глобальную область видимости, но не могут иметь локальную область видимости. (Функция не может быть определена внутри блока, поскольку если бы она могла иметь локальную область видимости, то была бы известна только самой себе и, следовательно, не могла быть вызванной из другой функции. Такая функция вообще не могла бы считаться функцией.)

Различные варианты хранения в C++ характеризуются продолжительностью существования, областью видимости и связыванием. Давайте рассмотрим классы хранения C++ в терминах их свойств. Начнем с исследования ситуации, имевшей место до ввода в язык пространств имен, и посмотрим, как они изменили общую картину.

Автоматическая продолжительность хранения

Параметры функции и переменные, объявленные внутри функции, по умолчанию имеют автоматическую продолжительность хранения. Они также обладают локальной областью видимости и не имеют связывания. Другими словами, если объявить переменную по имени `texas` в `main()`, а затем объявить еще одну переменную с тем же именем в функции `oil()`, будут созданы две независимые переменные, каждая из которых известна только в той функции, в которой объявлена. Любые операции с переменной `texas` в функции `oil()` не оказывают влияния на переменную `texas` в `main()` и наоборот. Кроме того, каждой переменной выделяется память, когда выполнение программы входит в самый вложенный блок, содержащий определение переменной, и каждая переменная прекращает существование, когда выполнение программы покидает этот блок. (Обратите внимание, что переменной выделяется память, когда выполнение программы входит в такой блок, но область видимости начинается только после точки объявления.)

Если определить переменную внутри блока, ее время существования и область видимости ограничиваются этим блоком. Предположим, что в начале `main()` определена переменная `teledeli`. Теперь пусть в `main()` создается новый блок, в котором определяется новая переменная по имени `websight`. В этом случае переменная `teledeli` является видимой как во внешнем, так и во внутреннем блоке, в то время как `websight` существует только во внутреннем блоке и находится в области видимости с точки своего определения до тех пор, пока выполнение программы не доберется до конца блока:

```
int main()
{
   int teledeli = 5;
   { // Переменной websight выделяется память
      cout << "Hello\n";
      int websight = -2;      // начинается область видимости websight
      cout << websight << ' ' << teledeli << endl;
   } // websight прекращает существование
   cout << teledeli << endl;
   ...
} // Переменная teledeli прекращает существование
```

А что если переменной во внутреннем блоке назначить имя `teledeli` вместо `websight`, в результате чего получится две переменных с одним и тем же именем, одна из которых находится во внешнем блоке, а другая — во внутреннем? В этом случае программа интерпретирует имя `teledeli` как переменную, локальную по отношению к блоку, во время выполнения операторов этого блока. Принято говорить, что новое определение *скрывает* предыдущее. Новое определение попадает в область видимости, а предыдущее из нее временно удаляется. Когда выполнение программы покидает блок, исходное определение возвращается обратно в область видимости (рис. 9.2).

Код в листинге 9.4 показывает, что автоматические переменные локализованы внутри функций или блоков, которые их содержат.

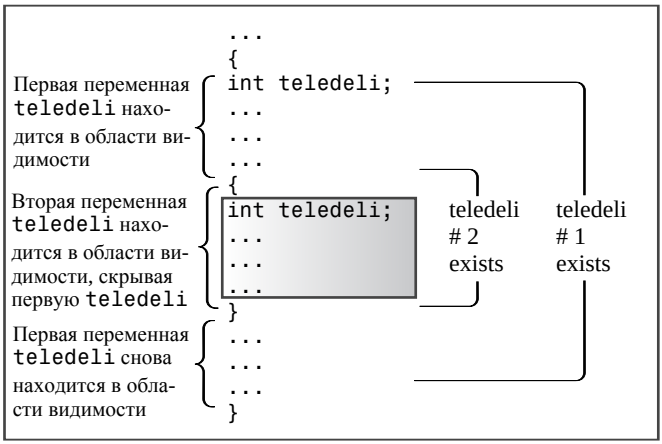


***Рис. 9.2.** Блоки и область видимости*

Листинг 9.4. `auto.cpp`

```
// auto.cpp -- иллюстрация области видимости автоматических переменных
#include <iostream>
void oil(int x);
int main()
{
   using namespace std;
   int texas = 31;
   int year = 2011;
   cout << "In main(), texas = " << texas << ", &texas = ";
   cout << &texas << endl;
   cout << "In main(), year = " << year << ", &year = ";
   cout << &year << endl;
   oil(texas);
   cout << "In main(), texas = " << texas << ", &texas = ";
   cout << &texas << endl;
   cout << "In main(), year = " << year << ", &year = ";
   cout << &year << endl;
   return 0;
}
void oil(int x)
{
   using namespace std;
   int texas = 5;
   cout << "In oil(), texas = " << texas << ", &texas = ";
   cout << &texas << endl;
   cout << "In oil(), x = " << x << ", &x = ";
   cout << &x << endl;
   {                        // начало блока
      int texas = 113;
      cout << "In block, texas = " << texas;
      cout << ", &texas = " << &texas << endl;
      cout << "In block, x = " << x << ", &x = ";
      cout << &x << endl;
   }                        // конец блока
   cout << "Post-block texas = " << texas;
   cout << ", &texas = " << &texas << endl;
}
```

Ниже показан вывод программы из листинга 9.4:

```
In main(), texas = 31, &texas = 0012FED4
In main(), year = 2011, &year = 0012FEC8
In oil(), texas = 5, &texas = 0012FDE4
In oil(), x = 31, &x = 0012FDF4
In block, texas = 113, &texas = 0012FDD8
In block, x = 31, &x = 0012FDF4
Post-block texas = 5, &texas = 0012FDE4
In main(), texas = 31, &texas = 0012FED4
In main(), year = 2011, &year = 0012FEC8
```

Обратите внимание, что каждая из трех переменных `texas` в листинге 9.4 имеет собственный, отличающийся от других адрес, и программа использует только ту переменную, которая в данное время находится в области видимости. Поэтому присваивание переменной `texas` значения `113` во внутреннем блоке функции `oil()` никак не отражается на других переменных с тем же именем. (Как обычно, конкретные значения адресов и формат представления варьируются от системы к системе.)

Рассмотрим последовательность событий. Когда `main()` начинается, программа выделяет память для переменных `texas` и `year`, и они обе попадают в область видимости. Когда программа вызывает функцию `oil()`, эти переменные остаются в памяти, но покидают область видимости. Две новых переменных, `x` и `texas`, размещаются в памяти и попадают в область видимости. Когда выполнение программы достигает внутреннего блока в функции `oil()`, новая переменная `texas` выходит из области видимости (скрывается), поскольку замещается более новым определением. Однако переменная `x` остается в области видимости, т.к. в блоке новая переменная с таким же именем не определяется. Когда выполнение программы выходит за пределы этого блока, освобождается память, занятая самой новой переменной `texas`, а вторая переменная `texas` возвращается в область видимости. После завершения функции `oil()` переменные `texas` и `x` перестают существовать, а в область видимости возвращаются исходные переменные `texas` и `year`.

Изменения, связанные с ключевым словом auto, в C++11

В C++11 ключевое слово `auto` используется для автоматического выведения типа, как уже было показано в главах 3, 7 и 8. Однако в C и предшествующих версиях C++ ключевое слово `auto` имеет совершенно другое предназначение. Оно служит для явного указания того, что переменная имеет автоматическое хранение:

```
int froob(int n)
{
   auto float ford;  // ford получает автоматическое хранение
   ...
}
```

Поскольку программисты могут использовать ключевое слово `auto` только с переменными, которые по умолчанию уже являются автоматическими, это слово применялось редко. Главное его назначение заключается в документировании того факта, что действительно требуется локальная автоматическая переменная.

В C++11 такое использование больше не является допустимым. Люди, занимающиеся подготовкой стандартов, неохотно изменяют назначение ключевых слов, потому что в результате может перестать работать код, в котором эти ключевые слова применялись для других целей. В данном случае было высказано мнение, что в прошлом слово `auto` использовалось настолько редко, что вполне допустимо перепрофилировать его, нежели вводить новое ключевое слово.

Инициализация автоматических переменных

Автоматическую переменную можно инициализировать с помощью любого выражения, значение которого известно на момент объявления переменной. Ниже приведен пример инициализации переменных x, big, y и z:

```
int w;                  // значение w не определено
int x = 5;              // инициализация числовым литералом
int big = INT_MAX - 1;  // инициализация константным выражением
int y = 2 * x;          // использование ранее определенного значения x
cin >> w;
int z = 3 * w;          // использование нового значения w
```

Автоматические переменные и стек

Чтобы получить более полное представление об автоматических переменных, рассмотрим их реализацию обычным компилятором C++. Поскольку количество автоматических переменных растет или сокращается по мере того, как функции начинают и завершают выполнение, программа должна управлять автоматическими переменными в процессе своей работы. Стандартная методика состоит в выделении области памяти, которая будет использоваться в качестве стека, управляющего движением переменных.

Термин *стек* применяется потому, что новые данные размещаются, образно говоря, поверх старых данных (т.е. в смежных, а не в тех же самых ячейках памяти), а затем удаляются из стека, после того как программа завершит работу с ними. По умолчанию размер стека зависит от реализации, однако обычно компилятор предоставляет опцию изменения размера стека.

Программа отслеживает состояние стека с помощью двух указателей. Один указывает на базу стека, с которой начинается выделенная область памяти, а другой — на вершину стека, которая представляет собой следующую ячейку свободной памяти. Когда происходит вызов функции, ее автоматические переменные добавляются в стек, а указатель вершины устанавливается на свободную ячейку памяти, следующую за только что размещенными переменными. После завершения функции указатель вершины снова принимает значение, которое он имел до вызова функции. В результате эффективно освобождается память, которая использовалась для хранения новых переменных.

Стек построен по принципу LIFO (last-in, first-out — последним пришел, первым обслужен). Это означает, что переменная, которая попала в стек последней, удаляется из него первой. Такой механизм упрощает передачу аргументов. Вызов функции помещает значения ее аргументов в вершину стека и переустанавливает указатель вершины. Вызванная функция использует описание своих формальных параметров для определения адреса каждого аргумента. Например, на рис. 9.3 показана функция `fib()`, которая в момент вызова передает двухбайтное значение типа `int` и четырехбайтное значение типа `long`. Эти значения помещаются в стек. Когда функция `fib()` начинает выполняться, она связывает имена `real` и `tell` с этими двумя значениями. После завершения работы функции `fib()` указатель вершины стека возвращается в прежнее состояние. Новые значения не удаляются, но теперь они лишаются меток, и пространство памяти, которое они занимают, будет использовано следующим процессом, в ходе которого будут размещаться значения в стеке. (На рис. 9.3 показана упрощенная картина, поскольку при вызове функции может передаваться дополнительная информация, такая как адрес возврата.)

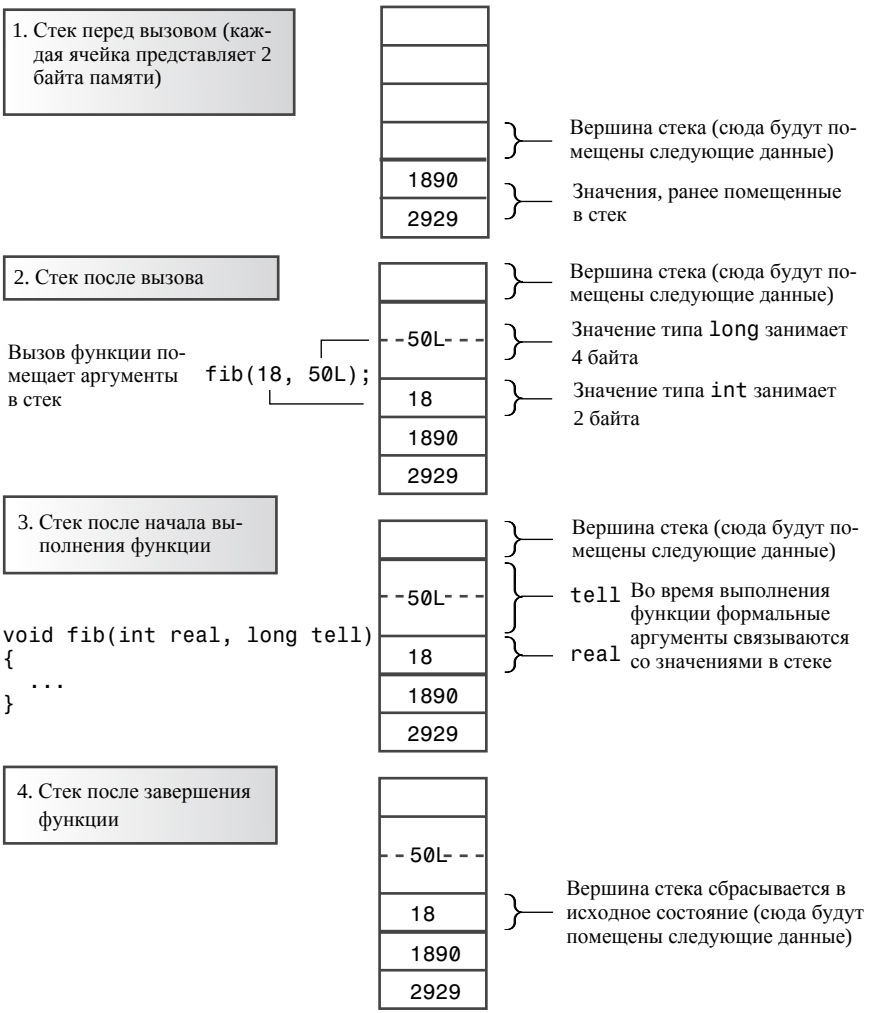


Рис. 9.3. *Передача аргументов с использованием стека*

Регистровые переменные

Ключевое слово `register` было первоначально введено в языке C, чтобы рекомендовать компилятору использовать для хранения автоматической переменной регистр центрального процессора:

```
register int count_fast;  // запрос на создание регистровой переменной
```

Идея заключалась в том, что это ускоряло доступ к переменной.

До появления стандарта C++11 это ключевое слово применялось в C++ похожим образом, но с одним отличием. Поскольку оборудование и компиляторы стали более совершенными, эта рекомендация была обобщена и начала указывать на тот факт, что переменная интенсивно используется, и, возможно, компилятор сумеет уделить ей особое внимание. В C++11 эта рекомендация является устаревшей и ключевое слово `register` остается просто способом идентифицировать переменную как автоматическую. Учитывая, что `register` может применяться только с переменными, которые будут автоматическими в любом случае, одна из причин использования этого ключево-

го слова — указать, что действительно нужна автоматическая переменная, возможно, с тем же самым именем, что и у внешней переменной. Точно таким же было первоначальное назначение auto. Однако более важная причина того, что ключевое слово register осталось, связана с желанием сохранить допустимым существующий код, в котором оно используется.

Переменные со статической продолжительностью хранения

Язык C++, как и C, предоставляет переменные со статической продолжительностью хранения с тремя видами связывания: внешнее (возможность доступа в разных файлах), внутреннее (возможность доступа к функциям внутри одного файла) и отсутствие связывания (возможность доступа только к одной функции или к одному блоку внутри функции). Переменные с этими тремя типами связывания существуют в течение всего времени выполнения программы; они долговечнее автоматических переменных. Поскольку количество статических переменных не меняется на протяжении выполнения программы, она не нуждается в специальных механизмах, подобных стеку, чтобы управлять ими. Компилятор просто резервирует фиксированный блок памяти для хранения всех статических переменных, и эти переменные доступны программе на протяжении всего времени ее выполнения. Более того, если статическая переменная не инициализирована явно, компилятор устанавливает ее в 0. Элементы статических массивов и структур устанавливаются в 0 по умолчанию.

На заметку!

Классический стандарт K&R C не позволяет инициализировать автоматические массивы и структуры, но допускает инициализацию статических массивов и структур. В ANSI C и C++ разрешено инициализировать обе разновидности данных. Однако некоторые ранние трансляторы C++ используют компиляторы языка C, которые не полностью совместимы со стандартом ANSI C. Если вы пользуетесь такой реализацией, то для инициализации массивов и структур может возникнуть необходимость воспользоваться одним из трех видов статических классов хранения.

Рассмотрим создание всех трех видов переменных со статической продолжительностью хранения, а затем приступим к исследованию их свойств. Чтобы создать статическую переменную с внешним связыванием, ее нужно объявить вне всех блоков. Чтобы создать статическую переменную с внутренним связыванием, ее следует объявить вне всех блоков и указать модификатор класса хранения static. Для создания статической переменной без связывания ее нужно объявить внутри какого-либо блока, используя модификатор static. В следующем фрагменте кода демонстрируются все три случая:

```
...
int global = 1000;       // статическая продолжительность, внешнее связывание
static int one_file = 50; // статическая продолжительность, внутреннее связывание
int main()
{
    ...

void funct1(int n)
{
    static int count = 0; // статическая продолжительность, нет связывания
    int llama = 0;
    ...
}
```

```
void funct2(int q)
{
   ...
}
```

Как уже упоминалось ранее, все переменные со статической продолжительностью хранения (в приведенном примере `global`, `one_file` и `count`) существуют с момента начала выполнения программы и до ее завершения. Переменная `count`, объявленная внутри функции `funct1()`, характеризуется локальной областью видимости и отсутствием связывания. Это означает, что она может использоваться только в рамках функции `funct1()`, точно так же как автоматическая переменная `llama`. Но, в отличие от `llama`, переменная `count` остается в памяти, даже когда функция `funct1()` не выполняется. Переменные `global` и `one_file` имеют область видимости файла, а это значит, что они могут использоваться, начиная с точки объявления и до конца файла. В частности, с обеими переменными можно работать в функциях `main()`, `funct1()` и `funct2()`. Так как переменная `one_file` имеет внутреннее связывание, она может использоваться только в файле, содержащем этот код. Поскольку переменная `global` имеет внешнее связывание, она также может применяться в других файлах, которые являются частью программы.

Все статические переменные обладают следующей особенностью инициализации: все биты неинициализированной статической переменной устанавливаются в 0. Такая переменная называется *инициализированной нулями*.

В табл. 9.1 приведена сводка по характеристикам классов хранения, которые были актуальны до появления пространств имен. Далее мы рассмотрим разновидности переменных со статической продолжительностью хранения более подробно.

Таблица 9.1. Пять видов хранения переменных

Описание хранения	Продолжительность	Область видимости	Связывание	Способ объявления
Автоматическая	Автоматическая	Блок	Нет	В блоке
Регистровая	Автоматическая	Блок	Нет	В блоке, с использованием ключевого слова `register`
Статическая без связывания	Статическая	Блок	Нет	В блоке, с использованием ключевого слова `static`
Статическая с внешним связыванием	Статическая	Файл	Внешнее	Вне всех функций
Статическая с внутренним связыванием	Статическая	Файл	Внутреннее	Вне всех функций, с использованием ключевого слова `static`

Обратите внимание, что в двух случаях упоминания в табл. 9.1 ключевое слово `static` имеет несколько отличающийся смысл. При использовании в локальном объявлении для указания статической переменной без связывания `static` отражает вид продолжительности хранения. Когда ключевое слово `static` применяется с объявлением вне блока, оно отражает внутреннее связывание; переменная уже имеет статическую продолжительность хранения. Это можно назвать *перегрузкой ключевого слова*, причем более точный смысл определяется контекстом.

Инициализация статических переменных

Статические переменные могут быть инициализированными нулями, они могут быть подвергнуты *инициализации константным выражением*, и они могут быть подвергнуты *динамической инициализации*. Как вы уже, наверное, догадались, инициализация нулями означает установку переменной в значение ноль. Для скалярных типов ноль предусматривает приведение к соответствующему типу. Например, нулевой указатель, который представлен как `0` в коде C++, может иметь ненулевое внутреннее представление, поэтому переменная типа указателя будет инициализирована этим значением. Члены структуры являются инициализированными нулями, и любой заполняющий бит установлен в ноль.

Инициализация нулями и инициализация константным выражением вместе называются *статической инициализацией*. Это значит, что переменная инициализируется, когда компилятор обрабатывает файл (или единицу трансляции). Динамическая инициализация означает, что переменная инициализируется позже.

Так что же определяет, какая форма инициализации будет применена? Прежде всего, все статические переменные являются инициализированными нулями, указана какая-либо инициализация или нет. Далее, если переменная инициализируется константным выражением, которое компилятор может вычислить исключительно на основе содержимого файла (учитывая включаемые заголовочные файлы), возможно проведение инициализации константным выражением. При необходимости компилятор готов выполнить простые вычисления. Если к этому моменту информации недостаточно, переменная будет инициализирована динамически.

Рассмотрим следующий код:

```
#include <cmath>
int x;                                  // инициализация нулями
int y = 5;                              // инициализация константным выражением
long z = 13 * 13;                       // инициализация константным выражением
const double pi = 4.0 * atan(1.0);      // динамическая инициализация
```

В начале переменные `x`, `y`, `z` и `pi` являются инициализированными нулями. Затем компилятор вычисляет константные выражения и инициализирует `y` и `z`, соответственно, значениями `5` и `169`. Но инициализация `pi` требует вызова функции `atan()`, и это должно подождать до тех пор, пока функция не будет скомпонована, а программа запущена.

Константное выражение не ограничено арифметическими выражениями, использующими литеральные константы. Например, в нем можно применять операцию `sizeof`:

```
int enough = 2 * sizeof (long) + 1; // инициализация константным выражением
```

В C++11 появилось новое ключевое слово `constexpr`, расширяющее возможности по созданию константных выражений; это одно из новых средств C++11, которые в настоящей книге не рассматриваются.

Статическая продолжительность хранения, внешнее связывание

Переменные с внешним связыванием часто называются просто *внешними переменными*. Они обязательно имеют статическую продолжительность хранения и область видимости файла. Внешние переменные определяются вне всех функций и поэтому являются внешними по отношению к любой функции. Например, они могут быть объявлены до описания функции `main()` или в заголовочном файле. Внешнюю переменную можно использовать в любой функции, которая следует в файле после определе-

ления переменной. Поэтому внешние переменные также называются *глобальными* – в отличие от автоматических переменных, которые являются локальными.

Правило одного определения

С одной стороны, внешняя переменная должна быть объявлена в каждом файле, в котором она будет использоваться. С другой стороны, в C++ имеется так называемое "правило одного определения" (one definition rule – odr), которое гласит, что для каждой переменной должно существовать только одно определение. Чтобы удовлетворить этим требованиям, в C++ доступно два вида объявления переменных. Первый вид называется *определяющим объявлением* или просто *определением*. Определение приводит к выделению памяти для переменной. Второй вид называется *ссылочным объявлением* или просто *объявлением*. Объявление не приводит к выделению памяти, поскольку ссылается на переменную, которая уже существует.

Ссылочное объявление использует ключевое слово `extern` и не предоставляет возможности инициализации. В противном случае объявление является определением и приводит к выделению пространства для хранения:

```
double up;           // определение, up равно 0
extern int blem;     // переменная blem определена в другом месте
extern char gr = 'z';// определение, поскольку присутствует инициализация
```

Если внешняя переменная используется в нескольких файлах, только один из них может содержать определение этой переменной (согласно правилу одного определения). Но во всех прочих файлах, где эта переменная используется, она должна быть объявлена с указанием ключевого слова `extern`:

```
// file01.cpp
extern int cats = 20;// определение, поскольку присутствует инициализация
int dogs = 22;       // тоже определение
int fleas;           // и это определение
...
// file02.cpp
// используются cats и dogs из file01.cpp
extern int cats;     // это не определения, поскольку в них указано
extern int dogs;     // ключевое слово extern и отсутствует инициализация
...
// file98.cpp
// используются cats, dogs и fleas из file01.cpp
extern int cats;
extern int dogs;
extern int fleas;
...
```

В этом случае во всех файлах используются переменные `cats` и `dogs`, определенные в `file01.cpp`. Однако в `file02.cpp` переменная `fleas` не объявляется повторно, поэтому доступ к ней невозможен. Ключевое слово `extern` в `file01.cpp` в действительности не нужно, поскольку и без него эффект будет таким же (рис. 9.4).

Обратите внимание, что правило одного определения не означает возможность существования только одной переменной с заданным именем. Например, автоматические переменные, разделяющие между собой одно и то же имя, но определенные в разных функциях, являются отдельными переменными, не зависящими друг от друга, и каждая из них обладает собственным адресом. Кроме того, как будет показано в последующих примерах, локальная переменная может скрывать глобальную переменную с тем же самым именем.

```
// программа file1.cpp
#include <iostream>
using namespace std;

// прототипы функций
#include "mystuff.h"

// определение внешней переменной
int process_status = 0;

void promise ();
int main()
{
   ...
}

void promise ()
{
   ...
}
```

В этом файле определяется переменная `process_status`, в результате чего компилятор выделяет для нее память.

```
// программа  file2.cpp
#include <iostream>
using namespace std;

// прототипы функций
#include "mystuff.h"

// ссылка на внешнюю переменную
extern int process_status;

int manipulate(int n)
{
   ...
}

char * remark(char * str)
{
   ...
}
```

В этом файле используется ключевое слово `extern`, которое указывает программе использовать переменную `process_status`, определенную в другом файле

***Рис. 9.4.** Определяющее объявление и ссылочное объявление*

Тем не менее, хотя в программе могут присутствовать различные переменные с одинаковыми именами, каждая версия может иметь только одно определение.

А что если определить внешнюю переменную и затем объявить обычную переменную с тем же самым именем внутри функции? Второе объявление интерпретируется как определение автоматической переменной. Эта автоматическая переменная находится в области видимости, когда программа выполняет эту конкретную функцию. Код в листингах 9.5 и 9.6 в случае совместной компиляции иллюстрирует использование внешней переменной в двух файлах и сокрытие глобальной переменной объявлением автоматической переменной с тем же именем. Программа также демонстрирует применение ключевого слова `extern` для повторного объявления внешней переменной, определенной ранее, а также использование операции разрешения контекста для реализации доступа к иначе скрытой внешней переменной.

Листинг 9.5. `external.cpp`

```
// external.cpp -- внешние переменные
// Компилировать вместе с support.cpp
#include <iostream>
using namespace std;
// Внешняя переменная
double warming = 0.3;        // переменная warming определена
// Прототипы функций
void update(double dt);
void local();
int main()                   // использует глобальную переменную
{
   cout << "Global warming is " << warming << " degrees.\n";
   update(0.1);              // вызов функции, изменяющей warming
   cout << "Global warming is " << warming << " degrees.\n";
   local();                  // вызов функции с локальной переменной warming
   cout << "Global warming is " << warming << " degrees.\n";
   return 0;
}
```

Листинг 9.6. support.cpp

```
// support.cpp -- использование внешних переменных
// Компилировать вместе с external.cpp
#include <iostream>
extern double warming;         // использование переменной warming из другого файла

// Прототипы функций
void update(double dt);
void local();

using std::cout;
void update(double dt)         // модифицирует глобальную переменную
{
   extern double warming;      // необязательное повторное объявление
   warming += dt;              // использование глобальной переменной warming
   cout << "Updating global warming to " << warming;
   cout << " degrees.\n";
}
void local()                   // использует локальную переменную
{
   double warming = 0.8;       // новая переменная скрывает внешнюю переменную
   cout << "Local warming = " << warming << " degrees.\n";
   // Доступ к глобальной переменной с помощью операции разрешения контекста
   cout << "But global warming = " << ::warming;
   cout << " degrees.\n";
}
```

Ниже показан вывод программы из листингов 9.5 и 9.6:

```
Global warming is 0.3 degrees.
Updating global warming to 0.4 degrees.
Global warming is 0.4 degrees.
Local warming = 0.8 degrees.
But global warming = 0.4 degrees.
Global warming is 0.4 degrees.
```

Замечания по программе

Вывод программы из листингов 9.5 и 9.6 показывает, что функции `main()` и `update()` имеют доступ к внешней переменной `warming`. Обратите внимание, что изменение, которое вносит функция `update()` в переменную `warming`, проявляется при последующих обращениях к этой переменной.

Определение переменной `warming` находится в листинге 9.5:

```
double warming = 0.3;   // переменная warming определена
```

В листинге 9.6 применяется ключевое слово `extern`, чтобы сделать переменную `warming` доступной функциям из этого файла:

```
extern double warming;  // использование переменной warming из другого файла
```

Это объявление означает: использовать переменную `warming`, определенную где-то во внешнем файле.

Вдобавок в функции `update()` осуществляется повторное объявление переменной `warming` за счет использования ключевого слова `extern`. Это ключевое слово означает: использовать переменную с таким именем, которая была внешне определена ранее. Поскольку функция `update()` будет работать и без этого объявления, оно является необязательным. Объявление призвано документировать, что данная функция предназначена для использования внешней переменной.

Функция `local()` показывает, что в случае объявления локальной переменной с тем же именем, что у глобальной, локальная переменная скрывает глобальную переменную. Например, функция `local()` при отображении значения `warming` использует локальное определение переменной `warming`.

Язык C++ расширяет возможности C за счет новой *операции разрешения контекста* (`::`). Если поместить эту операцию перед именем переменной, будет использоваться глобальная версия этой переменной. Таким образом, `local()` отображает для `warming` значение `0.8`, но для `::warming` — значение `0.4`. Эта операция еще будет неоднократно встречаться при обсуждении пространств имен и классов. Для обеспечения ясности и во избежание ошибок было бы лучше и безопаснее применять `::warming` в функции `update()` вместо просто `warming`, не полагаясь на правила области видимости.

Выбор между глобальными и локальными переменными

Теперь, когда имеется возможность выбора между глобальными и локальными переменными, возникает вопрос, каким из них отдать предпочтение? На первый взгляд глобальные переменные кажутся более привлекательными — поскольку все функции имеют к ним доступ, не нужно беспокоиться о передаче аргументов. Однако такой легкий доступ достается дорогой ценой — снижением надежности программ. Опыт показывает, что чем эффективнее программа изолирует данные от нежелательного доступа, тем лучше будет сохраняться их целостность. В большинстве случаев следует пользоваться локальными переменными и передавать данные функциям только по мере необходимости, а не делать данные открытыми за счет использования глобальных переменных. Как вы сможете убедиться позже, объектно-ориентированное программирование делает очередной шаг в плане изоляции данных.

Тем не менее, глобальные переменные имеют свою область применения. Предположим, что имеется блок данных, который должен использоваться несколькими функциями, такой как массив с названиями месяцев или список атомных весов химических элементов. Класс внешнего хранения наилучшим образом подходит для представления константных данных, поскольку в этом случае для предотвращения изменения данных можно воспользоваться ключевым словом `const`:

```
const char * const months[12] =
{
   "January", "February", "March", "April", "May",
   "June", "July", "August", "September", "October",
   "November", "December"
};
```

Первое ключевое слово `const` защищает от изменений строки, а второе слово `const` гарантирует, что каждый указатель в массиве будет постоянно указывать на ту же самую строку, на которую он указывал изначально.

Статическая продолжительность хранения, внутреннее связывание

Применение модификатора `static` к переменной с областью видимости файла обеспечивает для нее внутреннее связывание. Различие между внутренним и внешним связыванием становится значимым в многофайловых программах. В таком контексте переменная с внутренним связыванием является локальной для файла, который ее содержит. При этом обычная внешняя переменная обладает внешним связыванием, что означает возможность ее применения в различных файлах, как было показано в предыдущем примере.

А что если нужно использовать одно и то же имя для обозначения нескольких переменных в разных файлах? Можно ли просто опустить ключевое слово extern?

```
// файл 1
int errors = 20;           // внешнее объявление
...
-------------------------------------------
// файл 2
int errors = 5;            // ??известна только в file2??
void froobish()
{
   cout << errors;         // ошибка
   ...
```

Нет, это приведет к ошибке, потому что нарушается правило одного определения. Определение в файле 2 пытается создать внешнюю переменную, так что в программе оказывается два определения errors, что является ошибкой.

Однако если в файле объявляется статическая внешняя переменная с тем же именем, что и обычная внешняя переменная, объявленная в другом файле, то в область видимости первого файла попадает статическая версия:

```
// файл 1
int errors = 20;           // внешнее объявление
...
-------------------------------------------
// файл 2
static int errors = 5;     // известна только файлу 2
void froobish()
{
   cout << errors;         // использует переменную errors, определенную в файле 2
   ...
```

Это не нарушает правила одного определения, т.к. с помощью ключевого слова static для идентификатора errors обеспечивается внутреннее связывание, поэтому не предпринимается никаких попыток установить внешнее определение.

На заметку!

В многофайловой программе внешнюю переменную можно определять в одном и только одном файле. Все остальные файлы, использующие эту переменную, должны содержать ее объявление с ключевым словом extern.

Внешнюю переменную можно применять для разделения данных между различными частями многофайловой программы. Статическую переменную с внутренним связыванием можно использовать для разделения данных между различными функциями в одном файле. (Пространства имен предоставляют для этого альтернативный метод.) Кроме того, если переменную с областью видимости файла сделать static, то не придется беспокоиться о конфликте ее имени с переменными с областью видимости файла, которые содержатся в других файлах.

В листингах 9.7 и 9.8 показано, каким образом в C++ поддерживаются переменные с внешним и внутренним связыванием. В листинге 9.7 (twofile1.cpp) определены внешние переменные tom и dick, а также статическая внешняя переменная harry. Функция main() в этом файле отображает адреса всех трех переменных и затем вызывает функцию remote_access(), которая определена во втором файле. Содержимое этого файла (twofile2.cpp) приведено в листинге 9.8. Помимо определения функции remote_access(), в этом файле с помощью ключевого слова extern используется пе-

ременная tom из первого файла. Далее в нем определяется статическая переменная по имени dick. Модификатор static делает эту переменную локальной по отношению к файлу и переопределяет глобальное определение. Затем во втором файле определяется внешняя переменная по имени harry. При этом конфликт с переменной harry из первого файла не возникает, поскольку она обладает только внутренним связыванием. После этого функция remote_access() отображает адреса всех трех переменных, так что их можно сравнить с адресами соответствующих переменных из первого файла. Не забывайте, что для получения готовой программы потребуется скомпилировать и скомпоновать оба файла.

Листинг 9.7. twofile1.cpp

```
// twofile1.cpp -- переменные с внешним и внутренним связыванием
#include <iostream>       // должен компилироваться вместе с twofile2.cpp
int tom = 3;              // определение внешней переменной
int dick = 30;            // определение внешней переменной
static int harry = 300;   // статическая, внутреннее связывание

// Прототип функции
void remote_access();

int main()
{
   using namespace std;
   cout << "main() reports the following addresses:\n"; // вывод адресов
   cout << &tom << " = &tom, " << &dick << " = &dick, ";
   cout << &harry << " = &harry\n";
   remote_access();
   return 0;
}
```

Листинг 9.8. twofile2.cpp

```
// twofile2.cpp -- переменные с внутренним и внешним связыванием
#include <iostream>
extern int tom;           // переменная tom определена в другом месте
static int dick = 10;     // переопределяет внешнюю переменную dick
int harry = 200;          // определение внешней переменной,
                          // конфликт с harry из twofile1 отсутствует
void remote_access()
{
   using namespace std;
   cout << "remote_access() reports the following addresses:\n"; // вывод адресов
   cout << &tom << " = &tom, " << &dick << " = &dick, ";
   cout << &harry << " = &harry\n";
}
```

Ниже показан результат выполнения программы из листингов 9.7 и 9.8:

```
main() reports the following addresses:
0x0041a020 = &tom, 0x0041a024 = &dick, 0x0041a028 = &harry
remote_access() reports the following addresses:
0x0041a020 = &tom, 0x0041a450 = &dick, 0x0041a454 = &harry
```

По отображаемым адресам видно, что в обоих файлах используется одна и та же переменная tom, но разные переменные dick и harry. (Значения адресов и формат вывода зависят от системы, в которой выполняется программа. Тем не менее, адреса tom будут совпадать друг с другом, тогда как адреса dick и harry — отличаться.)

Статическая продолжительность хранения, отсутствие связывания

До сих пор мы рассматривали переменные, имеющие область видимости в пределах файла, с внешним и внутренним связыванием. Теперь обсудим третий член семейства со статической продолжительностью хранения – локальную переменную без связывания. Такая переменная создается за счет применения модификатора `static` к переменной, определенной внутри блока. Если она используется внутри блока, модификатор `static` задает локальной переменной статическую продолжительность хранения. Это означает, что, несмотря на видимость переменной в пределах блока, она существует даже тогда, когда блок неактивен. Таким образом, статическая локальная переменная может сохранять свое значение между вызовами функции. (Статические переменные полезны для реинкарнации – их можно применять для передачи секретных номеров счетов швейцарского банка вашему следующему воплощению.) Кроме того, если статическая локальная переменная инициализируется, это делается только один раз при запуске программы. Последующие вызовы функции не будут приводить к повторной инициализации переменной, как это происходит в случае автоматических переменных. Сказанное иллюстрируется в листинге 9.9.

Листинг 9.9. `static.cpp`

```
// static.cpp -- использование статической локальной переменной
#include <iostream>
// Константы
const int ArSize = 10;
// Прототип функции
void strcount(const char * str);

int main()
{
   using namespace std;
   char input[ArSize];
   char next;
   cout << "Enter a line: \n";
   cin.get(input, ArSize) ;
   while (cin)
   {
      cin.get(next);
      while (next != '\n')     // строка не помещается;
         cin.get(next);        // избавиться от остатка
      strcount(input);
      cout << "Enter next line (empty line to quit):\n";
      cin.get(input, ArSize);
   }
   cout << "Bye\n";
   return 0;
}
void strcount(const char * str)
{
   using namespace std;
   static int total = 0;       // статическая локальная переменная
   int count = 0;              // автоматическая локальная переменная
   cout <<  "\"" << str <<"\" contains ";
   while (*str++)              // переход к концу строки
      count++;
   total += count;
   cout << count << " characters\n";
   cout << total << " characters total\n";
}
```

Кстати, программа в листинге 9.9 демонстрирует один из способов обработки вводимой строки, которая может превышать размер выделенного для нее массива. Вспомните, что метод ввода `cin.get(input,ArSize)` читает до конца экранной строки или до позиции `ArSize-1`, в зависимости от того, что случится раньше. Символ новой строки остается во входной очереди. Программа использует метод `cin.get(next)` для чтения символа, который следует после введенной строки. Если `next` является символом новой строки, значит, предыдущий вызов `cin.get(input, ArSize)` должен был прочитать целиком всю строку. Если `next` не является символом новой строки, в строке ввода остались непрочитанные символы. Затем в программе с помощью цикла отбрасывается оставшаяся часть строки, но код можно изменить так, чтобы остаток строки был задействован в следующем цикле ввода. Кроме того, в программе используется тот факт, что попытка чтения пустой строки с помощью `get(char *, int)` приводит к тому, что `cin` возвращает `false`.

Ниже показан вывод программы из листинга 9.9:

```
Enter a line:
nice pants
"nice pant" contains 9 characters
9 characters total
Enter next line (empty line to quit):
thanks
"thanks" contains 6 characters
15 characters total
Enter next line (empty line to quit):
parting is such sweet sorrow
"parting i" contains 9 characters
24 characters total
Enter next line (empty line to quit):
ok
"ok" contains 2 characters
26 characters total
Enter next line (empty line to quit):

Bye
```

Обратите внимание, что поскольку размер массива равен `10`, программа не считывает более 9 символов на строку. Кроме того, автоматическая переменная `count` сбрасывается в `0` при каждом вызове функции. Однако статическая переменная `total` устанавливается в `0` только один раз в начале. После этого `total` сохраняет свое значение между вызовами функций, что позволяет ее использовать для подсчета текущей суммы.

Спецификаторы и классификаторы

Некоторые ключевые слова C++, называемые *спецификаторами класса хранения* и *cv-квалификаторами*, предоставляют дополнительную информацию о хранении. Ниже приведен список спецификаторов классов хранения:

- `auto` (исключен из спецификаторов в C++11)
- `register`
- `static`
- `extern`
- `thread_local` (добавлен в C++11)
- `mutable`

Большинство этих спецификаторов вы уже видели. В одном объявлении можно использовать не более одного из них, за исключением того, что `thread_local` может применяться со спецификаторами `static` и `extern`. Вспомните, что до появления C++11 ключевое слово `auto` могло использоваться в объявлении для документирования того факта, что переменная является автоматической. (В C++11 ключевое слово `auto` применяется для автоматического выведения типа.)

Ключевое слово `register` используется в объявлении для указания регистрового класса хранения, который в C++11 представляет собой всего лишь явный способ сообщения о том, что переменная является автоматической. Ключевое слово `static`, когда применяется в объявлении с областью видимости файла, задает внутреннее связывание. При использовании в локальном объявлении оно определяет статический класс хранения для локальной переменной. Ключевое слово `extern` указывает на ссылочное объявление — т.е., что объявление ссылается на переменную, которая определена где-то в другом месте. Ключевое слово `thread_local` отражает, что продолжительность хранения переменной является продолжительностью существования содержащего ее потока.

Переменная `thread_local` соотносится с потоком во многом так же, как обычная статическая переменная — со всей программой. Ключевое слово `mutable` объясняется в терминах `const`, поэтому давайте сначала рассмотрим cv-квалификаторы, а затем вернемся к `mutable`.

Cv-квалификаторы

Ниже перечислены cv-квалификаторы:

- `const`
- `volatile`

(Как и можно было догадаться, аббревиатура *cv* означает `const` и `volatile`.) Наиболее часто используемым cv-квалификатором является `const`, и вы уже видели его назначение: он указывает, что переменная после инициализации не может быть изменена программой. Чуть позже мы вернемся к обсуждению `const`.

Ключевое слово `volatile` указывает, что значение в ячейке памяти может быть изменено, даже если в коде программы нет ничего такого, что может модифицировать ее содержимое. Звучит загадочно, но все объясняется просто. Предположим, что имеется указатель на аппаратную ячейку памяти, в которой хранится время или информация, поступающая из порта. В этом случае содержимое изменяется оборудованием, а не программой. Или, например, две программы могут взаимодействовать, разделяя одни и те же данные. Назначение этого ключевого слова состоит в оптимизации возможностей компилятора. Предположим, компилятор обнаруживает, что программа использует значение некоторой переменной дважды в рамках нескольких операторов. Вместо того чтобы заставлять программу дважды обращаться к этому значению в хранилище, компилятор может кэшировать его в регистре. Такая оптимизация предполагает, что значение этой переменной не изменяется между двумя случаями ее использования. Если переменная не объявлена со спецификатором `volatile`, компилятор вправе предпринимать такую оптимизацию. Ключевое слово `volatile` говорит компилятору, что подобная оптимизация неприемлема.

`mutable`

Вернемся к спецификатору `mutable`. С его помощью можно указать, что отдельный член структуры (или класса) может быть изменен, даже если переменная типа структуры (или класса) объявлена со спецификатором `const`.

В качестве примера рассмотрим следующий код:

```
struct data
{
   char name[30];
   mutable int accesses;
   ...
};
const data veep = { "Claybourne Clodde", 0, ... };
strcpy(veep.name, "Joye Joux");    // не разрешено
veep.accesses++;                   // разрешено
```

Квалификатор `const` структуры `veep` предотвращает изменение ее элементов в программе, но спецификатор `mutable`, указанный для члена `accesses`, снимает с него это ограничение.

В настоящей книге спецификаторы `volatile` и `mutable` не используются, однако изучение `const` будет продолжено.

Дополнительные сведения о модификаторе `const`

В C++ (но не C) модификатор `const` привносит небольшие изменения в классы хранения, используемые по умолчанию. В то время как глобальная переменная по умолчанию обладает внешним связыванием, глобальная переменная со спецификатором `const` по умолчанию имеет внутреннее связывание. Другими словами, в C++ глобальное определение `const` обрабатывается так, будто в нем использован спецификатор `static`, как показано в следующем фрагменте кода:

```
const int fingers = 10;  // то же самое, что и static const int fingers = 10;
int main(void)
{
   ...
```

Чтобы упростить программирование, в C++ правила, регламентирующие использование константных типов, были несколько изменены. Предположим, что есть набор констант, которые требуется поместить в заголовочный файл. Этот файл будет использоваться в других файлах одной и той же программы. После того как препроцессор включит содержимое этого заголовочного файла в каждый исходный файл, во всех исходных файлах появятся следующие определения:

```
const int fingers = 10;
const char * warning = "Wak!";
```

Если бы глобальные объявления `const` имели внешнее связывание, как обычные переменные, это бы вызвало ошибку, поскольку нарушило бы правило одного определения. Это значит, что представленные выше объявления может содержать только один файл, в то время как в других файлах должны быть предусмотрены ссылочные объявления, использующие ключевое слово `extern`. Более того, только объявления без ключевого слова `extern` допускают инициализацию значений:

```
// Если бы у const было внешнее связывания,
// потребовалось бы ключевое слово extern
extern const int fingers;     // не может быть инициализирована
extern const char * warning;
```

Итак, потребовался бы один набор определений для одного файла и другой набор объявлений для остальных файлов. Однако поскольку определенные внешне данные `const` имеют внутреннее связывание, можно использовать одни и те же объявления во всех файлах.

Внутреннее связывание также означает, что каждый файл получает собственный набор констант, а не разделяет их с другими файлами. Каждое определение является приватным для файла, который его содержит. Именно поэтому определения констант целесообразно поместить в заголовочный файл. Таким образом, если включить один и тот же заголовочный файл в два файла исходного кода, они оба получат один и тот же набор констант.

Если по какой-либо причине необходимо, чтобы константа имела внешнее связывание, можно воспользоваться ключевым словом `extern` и переопределить устанавливаемое по умолчанию внутреннее связывание:

```
extern const int states = 50;  // определение с внешним связыванием
```

Для объявления константы во всех файлах, которые ее используют, должно применяться ключевое слово `extern`. В этом состоит отличие от обычной внешней переменной, при объявлении которой указывать ключевое слово `extern` не обязательно, но оно присутствует в остальных файлах, где данная переменная используется. Однако запомните, что теперь, когда одна константа может совместно использоваться множеством файлов, инициализировать ее разрешено только в одном файле.

При объявлении константы внутри функции или блока она получает область видимости блока. Это означает, что такая константа может применяться только во время выполнения кода данного блока. Это также означает возможность создания констант в функции или в блоке без риска возникновения конфликта имен с константами, определенными в других местах.

Функции и связывание

Подобно переменным, функции обладают свойствами связывания, хотя выбор в их случае более ограниченный. Язык C++, как и C, не позволяет объявлять одну функцию внутри другой, поэтому все функции автоматически получают статическую продолжительность хранения, т.е. существуют во время выполнения программы. По умолчанию функции имеют внешнее связывание, в том смысле, что могут разделяться между файлами. На самом деле в прототипе функции можно указать ключевое слово `extern`, отразив, что эта функция определена в другом файле, но это не обязательно. (Чтобы программа могла найти функцию в другом файле, этот файл должен быть одним из тех, который компилируется как часть программы, или библиотечным файлом, поиск которого осуществляет компоновщик.) Можно также ограничить область видимости функции одним файлом, назначив для нее внутреннее связывание с помощью ключевого слова `static`. Это ключевое слово должно применяться к прототипу и к определению функции:

```
static int private(double x);
...
static int private(double x)
{
    ...
}
```

В результате функция известна только в пределах данного файла. Это также означает, что то же самое имя можно назначить какой-то другой функции в другом файле. Как и в случае с переменными, статическая функция переопределяет внешнее определение для файла, содержащего статическое объявление. Таким образом, файл, содержащий определение статической функции, будет использовать именно эту версию функции даже при наличии внешнего объявления функции с таким же именем.

Правило одного определения распространяется также на невстроенные функции. Следовательно, каждая программа должна содержать в точности одно определение каждой невстроенной функции. Для функций с внешним связыванием это означает, что только один файл многофайловой программы может содержать определение функции. (Это может быть библиотечный файл, поставляемый кем-то другим.) Однако каждый файл, использующий функцию, должен содержать прототип этой функции.

Встроенные функции являются исключением из этого правила и позволяют помещать свои определения в заголовочный файл. Таким образом, каждый файл, который включает такой заголовочный файл, будет иметь определения встроенных функций. Однако язык C++ требует, чтобы все встроенные определения для конкретной функции были идентичными.

Где компилятор C++ ищет функции?

Предположим, что производится вызов функции в определенном файле программы. Где компилятор C++ будет искать определение этой функции? Если прототип функции в данном файле указывает, что функция является статической, компилятор ищет ее определение только в этом файле. В противном случае компилятор (а также и компоновщик) просматривает все файлы программы. Если компилятор находит два определения, он выдает сообщение об ошибке, поскольку может существовать только одно определение внешней функции. Если компилятору не удается обнаружить ни одного определения этой функции, он переходит к поиску в библиотеках. Отсюда вывод: если вы определите функцию с тем же именем, что и в библиотечной функции, компилятор будет использовать вашу версию функции, а не библиотечную. (Однако имена стандартных библиотечных функций в C++ зарезервированы, поэтому повторно использовать их нельзя.) Некоторые компиляторы-компоновщики для идентификации библиотек, где следует вести поиск, требуют указания явных инструкций.

Языковое связывание

Другая форма связывания, называемая *языковым связыванием*, касается функций. Начнем с основ. Для каждой отдельной функции компоновщику необходимо уникальное символическое имя. В C это реализуется просто, поскольку может существовать только одна функция с заданным именем. Поэтому для внутренних потребностей компилятор языка C может транслировать имя функции C, такое как `spiff`, в `_spiff`. Этот прием называется *языковым связыванием C*. Однако в C++ допускается наличие нескольких функций с одним и тем же именем, которые тоже должны транслироваться в разные символические имена. Таким образом, компилятор C++ инициирует процесс искажения или декорирования имен (рассмотренный в главе 8), позволяющий сгенерировать разные символические имена для перегруженных функций. Например, `spiff(int)` может быть преобразовано, скажем, в `_spiff_i`, а `spiff(double, double)` — в `_spiff_d_d`. Этот прием называется *языковым связыванием C++*.

Когда компоновщик ищет функцию C++, соответствующую вызову, он использует метод просмотра, отличный от метода, который применяется для поиска функции, соответствующей вызову C. Но предположим, что требуется использовать предварительно скомпилированную функцию из библиотеки C в программе на C++. Например, программа содержит следующий код:

```
spiff(22);  // обращение к функции spiff(int) из библиотеки C
```

Гипотетическое символическое имя в библиотеке C выглядит как `_spiff`, однако для нашего воображаемого компоновщика соглашение, принятое в отношении поиска в C++, диктует поиск символического имени `_spiff_i`.

Для решения этой проблемы можно воспользоваться прототипом функции, которое указывает, какой протокол следует применять:

```
extern "C" void spiff(int);    // использовать для поиска имени протокол C
extern void spoff(int);        // использовать для поиска имени протокол C++
extern "C++" void spaff(int); // использовать для поиска имени протокол C++
```

В первом примере используется языковое связывание C. Во втором и третьем примерах применяется языковое связывание C++; во втором примере это делается по умолчанию, а в третьем — явно.

Языковые связывания C и C++ являются единственными спецификаторами, которых требует стандарт C++. Реализации могут предоставлять дополнительные спецификаторы языкового связывания.

Схемы хранения и динамическое выделение памяти

Ранее было рассмотрено пять схем, исключая память в потоках, используемых в C++ для выделения памяти переменным (в том числе массивам и структурам). Они неприменимы к памяти, распределяемой с помощью операции `new` языка C++ (или более старой функции `malloc()` в C). Этот вид памяти называется *динамической памятью*. Как говорилось в главе 4, динамическая память управляется операциями `new` и `delete`, а не правилами, касающимися области видимости и связывания. Таким образом, динамическая память может выделяться в одной функции и освобождаться в другой. В отличие от автоматической памяти, динамическая память не подчиняется схеме LIFO. Порядок выделения и освобождения памяти зависит от того, когда и как применяются операции `new` и `delete`. Как правило, компилятор использует три отдельных области памяти: одну для статических переменных (эта область может быть разбита дополнительно), одну для автоматических переменных и одну для динамической памяти.

Несмотря на то что концепции схем хранения неприменимы к динамической памяти, они применимы к автоматическим и статическим переменным-указателям, используемым для отслеживания динамической памяти. Предположим, например, что функция содержит следующий оператор:

```
float * p_fees = new float [20];
```

80 байтов памяти (предполагая, что тип `float` занимает 4 байта), выделенных операцией `new`, остаются занятыми до тех пор, пока операция `delete` не освободит их. Однако указатель `p_fees` перестает существовать, когда выполнение программы покидает блок, содержащий его объявление. Если требуется, чтобы 80 байтов выделенной памяти стали доступными другой функции, необходимо передать или вернуть адрес этой памяти данной функции. С другой стороны, если сделать объявление указателя `p_fees` внешним, то он станет доступным всем функциям, которые находятся в файле после этого объявления. Используя следующий оператор во втором файле, можно сделать указатель доступным и в нем:

```
extern float * p_fees;
```

На заметку!

Выделяемая с помощью операции `new` память обычно освобождается после завершения работы программы. Однако так происходит не всегда. Например, в некоторых менее надежных операционных системах при определенных обстоятельствах запрос крупного блока памяти может приводить к ситуации, когда после завершения программы эта память не освобождается. Рекомендуемая практика предусматривает использование операции `delete` для освобождения памяти, выделенной с помощью `new`.

Инициализация с помощью операции new

А что если во время динамического выделения памяти нужно инициализировать переменную? В C++98 это было возможно в некоторых случаях. В C++11 спектр таких возможностей расширен. Давайте для начала посмотрим, что было доступно ранее.

Если требуется создать и инициализировать хранилище для одного из встроенных скалярных типов, таких как int или double, необходимо указать имя типа и инициализирующее значение, заключенное в круглых скобках:

```
int *pi = new int (6);                  // *pi устанавливается в 6
double * pd = new double (99.99);       // *pd устанавливается в 99.99
```

Синтаксис с круглыми скобками также может использоваться с классами, которые имеют подходящие конструкторы, но мы пока еще не добрались до этой темы.

Для инициализации обычной структуры или массива, однако, необходим стандарт C++11 и фигурные скобки списковой инициализации. Новый стандарт позволяет делать следующее:

```
struct where {double x; double y; double z;};
where * one = new where {2.5, 5.3, 7.2}; // C++11
int * ar = new int [4] {2,4,6,7};       // C++11
```

В C++11 можно также применять инициализацию с помощью фигурных скобок для переменных с одиночным значением:

```
int *pin = new int {});                 // *pi устанавливается в 6
double * pdo = new double {99.99};      // *pd устанавливается в 99.99
```

Когда new дает сбой

Может случиться так, что операция new не сможет найти запрошенный объем памяти. За первое десятилетие своего существования в C++ такая ситуация обрабатывалась возвратом нулевого указателя из операции new. Однако в настоящее время new генерирует исключение std::bad_alloc. В главе 15 приведено несколько коротких примеров, демонстрирующих каждый их подходов.

new: операции, функции и заменяющие функции

Операции new и new[] обращаются к двум функциям:

```
void * operator new(std::size_t);       // используется new
void * operator new[](std::size_t);     // используется new[]
```

Они называются *функциями распределения* и являются частью глобального пространства имен. Подобным же образом, существуют функции освобождения, вызываемые операциями delete и delete[]:

```
void operator delete(void *);
void operator delete[](void *);
```

Они используют синтаксис перегрузки операций, рассматриваемый в главе 11. Здесь std::size_t — это typedef для некоторых подходящих целочисленных типов. Базовый оператор, такой как

```
int * pi = new int;
```

транслируются примерно в следующее:

```
int * pi = new(sizeof(int));
```

А оператор

```
int * pa = new int[40];
```

транслируются в такую конструкцию:

```
int * pa = new(40 * sizeof(int));
```

Как видите, оператор с операцией new может также предоставлять инициализирующие значения, поэтому в общем случае использование new может сводиться не только к вызову функции new().

Аналогично, оператор

```
delete pi;
```

приводит к следующему вызову функции:

```
delete (pi);
```

Интересно то, что в C++ эти функции называются *заменяемыми*. Это значит, что при наличии достаточного опыта и желания можно создать заменяющие функции для new и delete, подогнав их под специфические требования. Например, можно было бы определить заменяющие функции с областью видимости класса и настроить их для удовлетворения потребностей в распределении конкретного класса. В коде операция new применялась бы как обычно, но вызывала бы заменяющую функцию new().

Операция new с размещением

Обычно операция new отвечает за поиск в куче блока памяти, который имеет достаточный размер, чтобы удовлетворить запрос памяти. Существует разновидность операции new, называемая *операцией new с размещением*, которая позволяет указывать адрес используемого блока. Программист может использовать это средство для построения собственных процедур управления памятью, для взаимодействия с оборудованием, доступ к которому осуществляется по определенному адресу, или для создания объектов в конкретной ячейке памяти.

Чтобы воспользоваться операцией new с размещением, сначала нужно включить заголовочный файл new, который содержит прототип этой версии new. Затем операция new применяется с аргументом, указывающим требуемый адрес. В остальном синтаксис операции new остается прежним. В частности, операция new с размещением может записываться со скобками или без них. В следующем фрагменте кода демонстрируется синтаксис всех четырех форм записи операции new:

```
#include <new>
struct chaff
{
   char dross[20];
   int slag;
};
char buffer1[50];
char buffer2[500];
int main()
{
   chaff *p1, *p2;
   int *p3, *p4;
   // Обычные формы операции new
   p1 = new chaff;       // помещение структуры в кучу
   p3 = new int[20];     // помещение массива int в кучу
```

```
// Две формы операции new с размещением
p2 = new (buffer1) chaff;      // помещение структуры в область buffer1
p4 = new (buffer2) int[20];    // помещение массива int в область buffer2
...
```

Ради простоты в этом примере для обеспечения пространством памяти операции new с размещением используются два статических массива. Таким образом, для структуры chaff выделяется область памяти buffer1, а для массива из 20 элементов int — область памяти buffer2.

Теперь, когда вы узнали, что такое операция new с размещением, рассмотрим пример программы. В листинге 9.10 оба вида операций new применяются для создания динамических массивов. В программе иллюстрируются некоторые важные различия между этими разновидностями new. Мы их обсудим после анализа вывода этой программы.

Листинг 9.10. newplace.cpp

```
// newplace.cpp -- использование операции new с размещением
#include <iostream>
#include <new>     // для операции new с размещением
const int BUF = 512;
const int N = 5;
char buffer[BUF];  // блок памяти
int main()
{
   using namespace std;
   double *pd1, *pd2;
   int i;
   // Вызов обычной и операции new с размещением
   cout << "Calling new and placement new:\n";
   pd1 = new double[N];                          // использование кучи
   pd2 = new (buffer) double[N];                 // использование массива buffer
   for (i = 0; i < N; i++)
      pd2[i] = pd1[i] = 1000 + 20.0 * i;
   cout << "Memory addresses:\n" << " heap: " << pd1
        << " static: " << (void *) buffer <<endl;// вывод адресов памяти
   cout << "Memory contents:\n";                 // вывод содержимого памяти
   for (i = 0; i < N; i++)
   {
      cout << pd1[i] << " at " << &pd1[i] << "; ";
      cout << pd2[i] << " at " << &pd2[i] << endl;
   }
   // Вызов обычной и операции new с размещением во второй раз
   cout << "\nCalling new and placement new a second time:\n";
   double *pd3, *pd4;
   pd3= new double[N];                           // нахождение нового адреса
   pd4 = new (buffer) double[N];                 // перезаписывание старых данных
   for (i = 0; i < N; i++)
      pd4[i] = pd3[i] = 1000 + 40.0 * i;
   cout << "Memory contents:\n";
   for (i = 0; i < N; i++)
   {
      cout << pd3[i] << " at " << &pd3[i] << "; ";
      cout << pd4[i] << " at " << &pd4[i] << endl;
   }
   // Вызов обычной и операции new с размещением в третий раз
   cout << "\nCalling new and placement new a third time:\n";
```

```
    delete [] pd1;
    pd1= new double[N];
    pd2 = new (buffer + N * sizeof(double)) double[N];
    for (i = 0; i < N; i++)
        pd2[i] = pd1[i] = 1000 + 60.0 * i;
    cout << "Memory contents:\n";
    for (i = 0; i < N; i++)
    {
        cout << pd1[i] << " at " << &pd1[i] << "; ";
        cout << pd2[i] << " at " << &pd2[i] << endl;
    }
    delete [] pd1;
    delete [] pd3;
    return 0;
}
```

Ниже показан вывод программы из листинга 9.10 на одной из систем:

```
Calling new and placement new:
Memory addresses:
heap: 006E4AB0 static: 00FD9138
Memory contents:
1000 at 006E4AB0; 1000 at 00FD9138
1020 at 006E4AB8; 1020 at 00FD9140
1040 at 006E4AC0; 1040 at 00FD9148
1060 at 006E4AC8; 1060 at 00FD9150
1080 at 006E4AD0; 1080 at 00FD9158

Calling new and placement new a second time:
Memory contents:
1000 at 006E4B68; 1000 at 00FD9138
1040 at 006E4B70; 1040 at 00FD9140
1080 at 006E4B78; 1080 at 00FD9148
1120 at 006E4B80; 1120 at 00FD9150
1160 at 006E4B88; 1160 at 00FD9158

Calling new and placement new a third time:
Memory contents:
1000 at 006E4AB0; 1000 at 00FD9160
1060 at 006E4AB8; 1060 at 00FD9168
1120 at 006E4AC0; 1120 at 00FD9170
1180 at 006E4AC8; 1180 at 00FD9178
1240 at 006E4AD0; 1240 at 00FD9180
```

Замечания по программе

Первое, что заслуживает внимания в листинге 9.10: операция `new` с размещением действительно помещает массив `p2` в массив `buffer`; обе переменные `p2` и `buffer` имеют значение `00FD9138`. Тем не менее, они относятся к разным типам; `p1` — это указатель на `double`, тогда как `buffer` — указатель на `char`. (Кстати, именно поэтому в программе используется приведение `(void *)` для `buffer`; в противном случае `cout` попытается отобразить строку.) Между тем, обычная операция `new` выделяет массиву `p1` адрес памяти с более высоким значением — `006E4AB0`, который принадлежит динамически управляемой куче.

Второй момент, который следует отметить, состоит в том, что второе обращение к обычной операции `new` приводит к нахождению другого блока памяти — начинающегося с адреса `006E4B68`. Однако второе обращение к операции `new` с размещением

приводит к использованию того же блока памяти, что и прежде. Этот блок начинается с адреса 00FD9138. Важный факт здесь в том, что операция new с размещением просто использует адрес, переданный в качестве аргумента; она не анализирует, свободна ли указанная область памяти, а также не ищет блок неиспользуемой памяти. В результате часть забот об управлении памятью возлагается на программиста. Например, при третьем обращении к операции new с размещением задается смещение в массиве buffer, чтобы использовалась новая область памяти:

```
pd2 = new (buffer + N * sizeof(double)) double[N]; // смещение на 40 байт
```

Третий момент касается наличия или отсутствия операции delete. Для обычной операции new следующий оператор освобождает блок памяти, начинающийся с адреса 006E4AB0; в результате следующее обращение к new может повторно использовать тот же самый блок:

```
delete [] pd1;
```

В противоположность этому программа из листинга 9.10 не использует delete для освобождения памяти, выделенной операцией new с размещением. На самом деле, в этом случае подобное невозможно. Область памяти, указанная переменной buffer, является статической, а delete может использоваться только с указателем на область памяти в куче, которая выделена обычной операцией new. Другими словами, массив buffer не подпадает под полномочия операции delete, поэтому следующий оператор вызовет ошибку времени выполнения:

```
delete [] pd2;  // не работает
```

С другой стороны, если бы для создания буфера памяти применялась обычная операция new, для освобождения всего блока памяти понадобилось бы обратиться к операции delete.

Использовать операцию new с размещением можно и другим способом — комбинировать ее с инициализацией для помещения информации по определенному аппаратному адресу.

Вас может заинтересовать, что конкретно делает операция new с размещением. В основном она лишь возвращает переданный ей адрес, приводя его к типу void *, чтобы его можно было присваивать любому типу указателя. Однако так работает стандартная операция new с размещением. C++ позволяет программистам перегружать эту операцию.

Ситуация усложняется, когда операция new с размещением используется для объектов классов. Эта тема будет продолжена в главе 12.

Другие формы операции new с размещением

Точно так же как обычная операция new вызывает функцию new с одним аргументом, стандартная операция new с размещением вызывает функцию new с двумя аргументами:

```
int * pi = new int;                 // вызывает new(sizeof(int))
int * p2 = new(buffer) int;         // вызывает new(sizeof(int), buffer)
int * p3 = new(buffer) int[40];     // вызывает new(40*sizeof(int), buffer)
```

Функция new с размещением не является заменяемой, но может быть перегружена. Ей необходимо передать, по крайней мере, два параметра, первым из которых всегда будет std::size_t, обозначающий количество запрашиваемых байт. Любая такая перегруженная функция называется функцией new с размещением, даже если дополнительные параметры не указывают ячейку памяти.

Пространства имен

Имена в C++ могут относиться к переменным, функциям, структурам, перечислениям, классам, а также членам классов и структур. По мере увеличения размеров программных проектов возрастает вероятность конфликта имен. При использовании библиотек классов из нескольких источников могут возникать конфликты имен. Например, две библиотеки могут определять классы с именами `List`, `Tree` и `Node`, но несовместимым образом. Может потребоваться использовать класс `List` из одной библиотеки и класс `Tree` из другой, при этом каждый из них ожидает взаимодействия с собственной версией класса `Node`. Конфликты подобного рода называются *проблемами пространств имен*.

Стандарт C++ предоставляет средства пространств имен, которые обеспечивают более совершенное управление областью видимости имен. Реализация средства пространств имен в компиляторах потребовало некоторого времени, но сейчас эта поддержка распространена повсеместно.

Традиционные пространства имен C++

Прежде чем приступать к изучению новых средств пространств имен, давайте ознакомимся со свойствами пространства имен, которые уже существуют в C++, и введем некоторые термины. Это поможет лучше освоиться с идеей, положенной в основу пространств имен.

Первый термин, с которым необходимо ознакомиться – это *декларативная область*. Декларативная область – это область, в которой могут делаться объявления. Например, глобальную переменную можно объявить вне всех функций. Декларативной областью для этой переменной является файл, в котором она объявлена. Если объявить переменную внутри функции, ее декларативной областью будет самый внутренний блок, в котором она объявлена.

Второй термин, который следует знать – это *потенциальная область видимости*. Потенциальная область видимости переменной начинается с точки объявления и простирается до конца ее декларативной области объявления. Таким образом, потенциальная область видимости более ограничена, чем декларативная область, поскольку нельзя использовать переменную в программе ранее позиции, в которой она впервые была определена.

Однако переменная может оказаться видимой не в каждом месте потенциальной области видимости. Например, она может быть скрыта другой переменной с тем же именем, объявленной во вложенной декларативной области. Скажем, локальная переменная, объявленная в функции (ее декларативной областью служит функция), скрывает глобальную переменную, объявленную в том же файле (ее декларативной областью является файл).

Часть программы, которой фактически доступна данная переменная, называется *областью видимости*, и именно в этом смысле данный термин применялся до сих пор. На рис. 9.5 и 9.6 проиллюстрированы термины *декларативная область*, *потенциальная область видимости* и *область видимости*.

Правила языка C++, касающиеся глобальных и локальных переменных, определяют своего рода иерархию пространств имен. В каждой декларативной области могут быть определены имена, не зависящие от имен, объявленных в других декларативных областях. Локальная переменная, объявленная в одной функции, не конфликтует с локальной переменной, объявленной в другой функции.

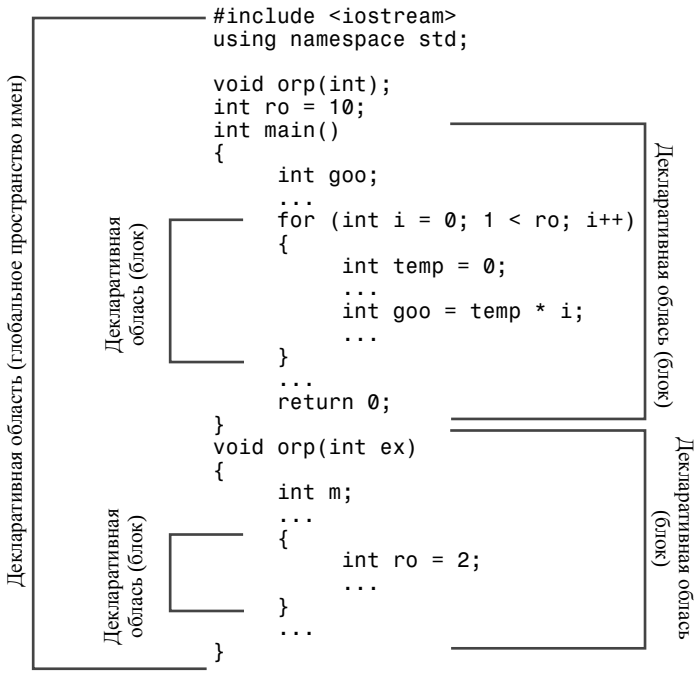


Рис. 9.5. *Декларативные области*

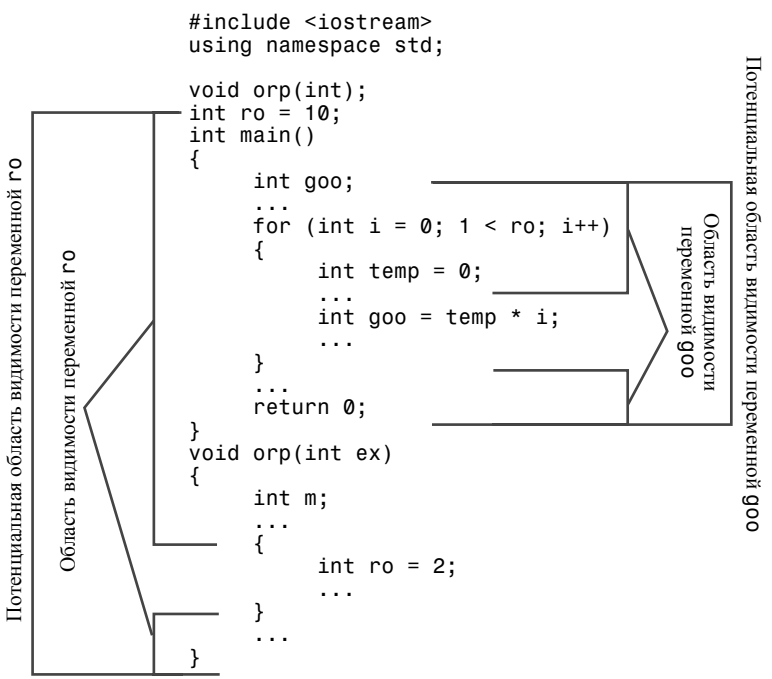


Рис. 9.6. *Потенциальная область видимости и область видимости*

Новое средство пространств имен

В настоящее время в C++ появилась возможность создавать именованные пространства имен за счет определения декларативной области нового вида, одним из основных назначений которой является предоставление места для объявления имен. Имена в одном пространстве имен не конфликтуют с такими же именами, но объявленными в других пространствах имен. При этом существуют механизмы, которые позволяют другим частям программы использовать элементы, объявленные в том или ином пространстве имен. Например, в приведенном ниже коде с помощью нового ключевого слова `namespace` создаются два пространства имен — `Jack` и `Jill`:

```
namespace Jack {
   double pail;                      // объявление переменной
   void fetch();                     // прототип функции
   int pal;                          // объявление переменной
   struct Well { ... };              // объявление структуры
}
namespace Jill {
   double bucket(double n) { ... } // определение функции
   double fetch;                     // объявление переменной
   int pal;                          // объявление переменной
   struct Hill { ... };              // объявление структуры
}
```

Пространства имен могут находиться на глобальном уровне или внутри других пространств имен, однако они не могут быть помещены в блок. Следовательно, имя, объявленное в пространстве имен, по умолчанию имеет внешнее связывание (если только оно не ссылается на константу).

В дополнение к пространствам имен, определяемым пользователем, существует еще одно пространство имен — *глобальное*. Оно соответствует декларативной области на уровне файла. Следовательно, то, что раньше подразумевалось под *глобальными переменными*, сейчас описывается как часть глобального пространства имен.

Имена в одном пространстве имен не конфликтуют с именами в другом пространстве имен. Таким образом, имя `fetch` в пространстве имен `Jack` может сосуществовать с именем `fetch` в пространстве `Jill`, а имя `Hill` пространства `Jill` — сосуществовать с внешним именем `Hill`. Правила, регламентирующие объявления и определения в пространствах имен, совпадают с правилами глобальных объявлений и определений.

Пространства имен являются *открытыми*. Это означает, что можно добавлять новые имена в существующие пространства имен. Например, следующий оператор добавляет имя `goose` к существующему списку имен пространства `Jill`:

```
namespace Jill {
   char * goose(const char *);
}
```

Подобным образом исходное пространство имен `Jack` предоставляет прототип функции `fetch()`. Код этой функции можно разместить далее в этом (или другом) файле, снова указав пространство имен `Jack`:

```
namespace Jack {
   void fetch()
   {
      ...
   }
}
```

Разумеется, для этого необходим какой-то способ доступа к именам заданного пространства имен. Простейший способ предусматривает использование операции разрешения контекста (::), которая позволяет *уточнить* имя с помощью его пространства имен:

```
Jack::pail = 12.34;  // использование переменной
Jill::Hill mole;     // создание структуры типа Hill
Jack::fetch();       // использование функции
```

Имя без добавлений, такое как pail, называется *не уточненным именем*, в то время как имя с указанием пространства имен вроде Jack::pail — *уточненным именем*.

Объявления using и директивы using

Необходимость в уточнении имен всякий раз, когда они используются — не особенно привлекательная перспектива, поэтому для упрощения использования имен некоторого пространства в C++ предлагаются два механизма — *объявление using* и *директива using*. Объявление using обеспечивает доступ к отдельным идентификаторам, а директива using делает доступным пространство имен в целом.

Объявление using предусматривает помещение ключевого слова using перед уточненным именем:

```
using Jill::fetch;        // объявление using
```

Объявление using добавляет в декларативную область отдельное имя. Например, объявление using для Jill::fetch в main() добавляет имя fetch в декларативную область, определенную функцией main(). После такого объявления имя fetch можно использовать вместо Jill::fetch. Сказанное иллюстрирует следующий фрагмент кода:

```
namespace Jill {
   double bucket(double n) { ... }
   double fetch;
   struct Hill { ... } ;
}
char fetch;
int main()
{
   using Jill::fetch;     // помещение fetch в локальное пространство имен
   double fetch;          // Ошибка! Локальное имя fetch уже существует!
   cin >> fetch;          // чтение значения в переменную Jill::fetch
   cin >> ::fetch;        // чтение значения в глобальную переменную fetch
   ...
}
```

Поскольку объявление using добавляет имя в локальную декларативную область, создание другой переменной с именем fetch в этом примере невозможно. Кроме того, как и любая другая локальная переменная, fetch переопределяет глобальную переменную с тем же самым именем.

Помещение объявления using на внешний уровень приводит к добавлению соответствующего имени в глобальное пространство имен:

```
void other();
namespace Jill {
   double bucket(double n) { ... }
   double fetch;
   struct Hill { ... };
}
```

```
using Jill::fetch;          // помещение fetch в глобальное пространство имен
int main()
{
   cin >> fetch;            // чтение значения в Jill::fetch
   other();
   ...
}
void other()
{
   cout << fetch;           // вывод значения Jill::fetch
   ...
}
```

Таким образом, объявление `using` делает доступным одиночное имя. В отличие от этого, директива `using` делает доступными все имена. Директива `using` создается путем предварения идентификатора пространства имен ключевыми словами `using namespace`. После этого *все* имена данного пространства становятся доступными без необходимости в использовании операции разрешения контекста:

```
using namespace Jack;       // делает доступными все имена в Jack
```

При размещении директивы на глобальном уровне имена пространства имен становятся доступными глобально. В этой книге данный прием демонстрировался несколько раз:

```
#include <iostream>         // помещает имена в пространство имен std
using namespace std;        // делает имена доступными глобально
```

Если поместить директиву `using` в отдельную функцию, имена станут доступными только в этой функции. Рассмотрим пример:

```
int vorn(int m)
{
   using namespace jack; // делает имена доступными в функции vorn()
   ...
}
```

Подобная форма уже неоднократно встречалась применительно к пространству имен `std`.

Необходимо учитывать, что директивы и объявления `using` увеличивают вероятность конфликта имен. Это значит, что если доступны пространства имен `jack` и `jill`, то в случае применения операции разрешения контекста неопределенность не возникает:

```
jack::pal = 3;
jill::pal = 10;
```

Переменные `jack::pal` и `jill::pal` имеют отличающиеся идентификаторы, относящиеся к разным адресам памяти. Однако при использовании объявлений `using` ситуация меняется:

```
using jack::pal;
using jill::pal;
pal = 4;          // какая переменная pal имеется в виду? возникает конфликт
```

В действительности компиляторы не позволяют указывать сразу оба таких объявления `using` по причине возникающей из-за этого неопределенности.

Сравнение директив `using` и объявлений `using`

Применение директивы `using` для импорта всех идентификаторов из пространства имен *не равнозначно* использованию множества объявлений `using`. Это больше напоминает массовое применение операции разрешения контекста. Использование объявления `using` равносильно ситуации, когда имя объявлено в области действия объявления `using`. Если некоторое имя уже объявлено в функции, нельзя импортировать то же самое имя с помощью объявления `using`. Однако в случае использования директивы `using` имеет место разрешение имен, как если бы соответствующие имена были объявлены в наименьшей декларативной области, содержащей как объявление `using`, так и само пространство имен.

В рассматриваемом ниже примере это будет глобальное пространство имен. Если воспользоваться директивой `using` для импорта некоторого имени, уже объявленного в функции, локальное имя будет скрывать имя из пространства имен точно так же, как оно скрывало бы глобальную переменную с тем же самым именем. Однако это не мешает применять операцию разрешения контекста, как показано в следующем примере:

```
namespace Jill {
   double bucket(double n) { ... }
   double fetch;
   struct Hill { ... };
}

char fetch;                    // глобальное пространство имен
int main()
{
   using namespace Jill;       // импорт всех имен из пространства
   Hill Thrill;                // создание структуры типа Jill::Hill
   double water = bucket(2);   // использование функции Jill::bucket();
   double fetch;               // это не ошибка; имя Jill::fetch скрывается
   cin >> fetch;               // чтение значения в локальную переменную fetch
   cin >> ::fetch;             // чтение значения в глобальную переменную fetch
   cin >> Jill::fetch;         // чтение значения в переменную Jill::fetch
   ...
}

int foom()
{
   Hill top;                   // ОШИБКА
   Jill::Hill crest;           // допустимо
}
```

Здесь в функции `main()` имя `Jill::fetch` помещено в локальное пространство имен. Оно не имеет локальной области видимости, следовательно, не переопределяет глобальное имя `fetch`. Однако локально объявленное имя `fetch` скрывает переменную `Jill::fetch` и глобальную переменную `fetch`. Тем не менее, обе эти переменные становятся доступными, если воспользоваться операцией разрешения контекста. Сравните этот пример с предыдущим, в котором применяется объявление `using`.

Еще следует отметить, что хотя директива `using` в функции рассматривает имена из области имен как объявленные вне функции, она не делает их доступными другим функциям в файле. Поэтому в предыдущем примере функция `foom()` не может использовать не уточненный идентификатор `Hill`.

На заметку!

Предположим, что одно и то же имя определено как в пространстве имен, так и в декларативной области. При попытке применить объявление `using`, чтобы перенести имя из пространства имен в декларативную область, два имени вступят в конфликт, и отобразится сообщение об ошибке. Если с помощью директивы `using` перенести имя из пространства имен в декларативную область, локальная версия этого имени скроет версию из пространства имен.

Вообще говоря, применение объявления `using` более безопасно, поскольку оно показывает только те имена, которые решено сделать доступными. И если такое имя конфликтует с локальным именем, компилятор сообщит об этом. Директива `using` добавляет все имена без исключений, даже те, которые могут быть не нужны. Если локальное имя вступает в конфликт, оно переопределяет версию имени из пространства имен, причем никаких предупреждений об этом не выводится. Кроме того, открытая природа пространств имен означает, что полный список имен некоторого пространства может распространяться на несколько местоположений, что усложняет задачу точного выяснения, какие имена были добавлены.

В большинстве примеров этой книги используется следующий подход:

```
#include <iostream>
int main()
{
   using namespace std;
```

Заголовочный файл `iostream` помещает все идентификаторы в пространство имен `std`. Затем директива `using` делает имена доступными внутри функции `main()`. В некоторых примерах применяется другой подход:

```
#include <iostream>
using namespace std;
int main()
{
```

Здесь все элементы пространства имен `std` экспортируются в глобальное пространство имен. Основное преимущество такого подхода состоит в рациональности. Его легко реализовать, к тому же, если используемая система не поддерживает пространства имен, первые две строки кода можно заменить исходной формой:

```
#include <iostream.h>
```

Однако ожидания сторонников пространств имен основаны на том, что пользователи будут более разборчивыми и воспользуются либо операцией разрешения контекста, либо объявлением `using`. Другими словами, по их мнению, не следует применять такое объявление:

```
using namespace std; // избегайте, поскольку конструкция слишком неразборчива
```

Вместо этого рекомендуется пользоваться следующим подходом:

```
int x;
std::cin >> x;
std::cout << x << std::endl;
```

Или поступать так:

```
using std::cin;
using std::cout;
using std::endl;
int x;
cin >> x;
cout << x << endl;
```

С помощью вложенных пространств имен, рассматриваемых в следующем разделе, можно создать пространство имен, содержащее часто используемые объявления using.

Другие возможности пространств имен

Объявления пространств имен могут быть вложенными, как показано ниже:

```
namespace elements
{
   namespace fire
   {
      int flame;
      ...
   }
   float water;
}
```

В этом случае ссылка на переменную flame выглядит как elements::fire::flame. Аналогичным образом внутренние имена можно сделать доступными с помощью следующей директивы using:

```
using namespace elements::fire;
```

Можно также пользоваться директивами и объявлениями using внутри пространств имен, как показано ниже:

```
namespace myth
{
   using Jill::fetch;
   using namespace elements;
   using std::cout;
   using std::cin;
}
```

Предположим, что требуется доступ к переменной Jill::fetch. Поскольку переменная Jill::fetch сейчас является частью пространства имен myth, в котором к ней можно обращаться по имени fetch, для доступа к переменной возможен следующий способ:

```
std::cin >> myth::fetch;
```

Разумеется, поскольку переменная является еще и частью пространства имен Jill, обращение Jill::fetch также допустимо:

```
std::cout << Jill:fetch; // вывод значения, прочитанного в myth::fetch
```

При условии отсутствия конфликта имен локальных переменных допустим и следующий вариант:

```
using namespace myth;
cin >> fetch;              // в действительности это std::cin и Jill::fetch
```

Теперь рассмотрим возможность применения директивы using в пространстве имен myth. Директива using является *транзитивной*. Считается, что операция *op* транзитивна, если из выражений A *op* B и B *op* C следует A *op* C. Например, операция > обладает свойством транзитивности. (Иначе говоря, из того, что A больше B и B больше C, следует, что A больше C.)

В данном контексте это означает, что выполнение приведенного ниже оператора приводит к помещению пространств имен myth и elements в область видимости:

```
using namespace myth;
```

Эта единственная директива имеет такой же эффект, что и следующие две директивы:

```
using namespace myth;
using namespace elements;
```

Для пространства имен можно создать псевдоним. Предположим, что имеется пространство имен, определенное так:

```
namespace my_very_favorite_things { ... };
```

Ниже показано, как сделать имя mvft псевдонимом для my_very_favorite_things:

```
namespace mvft = my_very_favorite_things;
```

Этот же прием можно применить для упрощения работы с вложенными пространствами имен:

```
namespace MEF = myth::elements::fire;
using MEF::flame;
```

Неименованные пространства имен

Для создания неименованного пространства имен необходимо опустить идентификатор после ключевого слова namespace:

```
namespace            // неименованное пространство имен
{
   int ice;
   int bandycoot;
}
```

Результат этих объявлений будет таким же, как если бы за ними следовала директива using. Другими словами, имена, объявленные в этом пространстве имен, находятся в потенциальной области видимости, которая продолжается до границы декларативной области, содержащей неименованное пространство имен. В этом отношении имена неименованного пространства подобны глобальным переменным. Однако если пространство не имеет имени, нельзя явно применить директиву или объявление using, чтобы сделать имена доступными в другом месте. В частности, идентификаторы из неименованного пространства имен нельзя использовать нигде, кроме файла, содержащего объявление этого пространства имен. Это может служить альтернативой применению статических переменных с внутренним связыванием. Рассмотрим следующий пример кода:

```
static int counts;   // статическая переменная, внутреннее связывание
int other;
int main()
{
   ...
}
int other()
{
   ...
}
```

С использованием пространств имен этот код можно переписать так:

```
namespace
{
   int counts;  // статическая переменная, внутреннее связывание
}
int other();
int main()
{
   ...
}
int other()
{
   ...
}
```

Пример пространства имен

Рассмотрим пример многофайловой программы, демонстрирующей некоторые возможности пространства имен. Первый файл (листинг 9.11) является заголовочным и содержит типичные для таких файлов элементы — константы, определения структур и прототипы функций. В этом случае элементы помещены в два пространства имен. Первое пространство имен, `pers`, содержит определение структуры `Person`, а также прототипы функции, которая помещает в структуру имя некоторого лица, и функции, отображающей содержимое структуры. Второе пространство имен, `debts`, определяет структуру для хранения имени лица и суммы его задолженности. Эта структура использует структуру `Person`, поэтому пространство имен `debts` содержит директиву `using`, которая делает имена пространства `pers` доступными в `debts`. Пространство имен `debts` также содержит ряд прототипов.

Листинг 9.11. `namesp.h`

```
// namesp.h
#include <string>
// Создание пространств имен pers и debts
namespace pers
{
   struct Person
   {
      std::string fname;
      std::string lname;
   };
   void getPerson(Person &);
   void showPerson(const Person &);
}
namespace debts
{
   using namespace pers;
   struct Debt
   {
      Person name;
      double amount;
   };
   void getDebt(Debt &);
   void showDebt(const Debt &);
   double sumDebts(const Debt ar[], int n);
}
```

Второй файл этого примера (листинг 9.12) следует обычному шаблону, при котором файл исходного кода содержит определения для прототипов функций из заголовочного файла. Имена функций, объявленные в пространстве имен, имеют область видимости пространства имен, поэтому определения должны находиться в том же самом пространстве имен, что и объявления. Это тот случай, когда открытая природа пространства имен становится удобной. Исходные пространства имен включаются с помощью файла `namesp.h` (листинг 9.11). Затем файл добавляет определения функций к двум пространствам имен, как показано в листинге 9.12. Кроме того, в файле `namesp.cpp` демонстрируется обеспечение доступа к элементам пространства имен `std` с помощью объявления `using` и операции разрешения контекста.

Листинг 9.12. `namesp.cpp`

```
// namesp.cpp -- пространства имен
#include <iostream>
#include "namesp.h"

namespace pers
{
   using std::cout;
   using std::cin;
   void getPerson(Person & rp)
   {
      cout << "Enter first name: ";      // ввод имени
      cin >> rp.fname;
      cout << "Enter last name: ";       // ввод фамилии
      cin >> rp.lname;
   }

   void showPerson(const Person & rp)
   {
      std::cout << rp.lname << ", " << rp.fname;
   }
}

namespace debts
{
   void getDebt(Debt & rd)
   {
      getPerson(rd.name);
      std::cout << "Enter debt: ";       // ввод суммы задолженности
      std::cin >> rd.amount;
   }

   void showDebt(const Debt & rd)
   {
      showPerson(rd.name);
      std::cout <<": $" << rd.amount << std::endl;
   }

   double sumDebts(const Debt ar[], int n)
   {
      double total = 0;
      for (int i = 0; i < n; i++)
         total += ar[i].amount;
      return total;
   }
}
```

Наконец, третий файл программы (листинг 9.13) представляет собой файл исходного кода, который использует структуры и функции, объявленные и определенные в пространствах имен. В листинге 9.13 показаны методы обеспечения доступа к идентификаторам пространства имен.

Листинг 9.13. usenmsp.cpp

```
// usenmsp.cpp -- использование пространств имен
#include <iostream>
#include "namesp.h"
void other(void);
void another(void);
int main(void)
{
   using debts::Debt;
   using debts::showDebt;
   Debt golf = { {"Benny", "Goatsniff"}, 120.0 };
   showDebt(golf);
   other();
   another();
   return 0;
}

void other(void)
{
   using std::cout;
   using std::endl;
   using namespace debts;
   Person dg = {"Doodles", "Glister"};
   showPerson(dg);
   cout << endl;
   Debt zippy[3];
   int i;
   for (i = 0; i < 3; i++)
      getDebt(zippy[i]);
   for (i = 0; i < 3; i++)
      showDebt(zippy[i]);
   cout << "Total debt: $" << sumDebts(zippy, 3) << endl;
   return;
}

void another(void)
{
   using pers::Person;
   Person collector = { "Milo", "Rightshift" };
   pers::showPerson(collector);
   std::cout << std::endl;
}
```

Функция `main()` в листинге 9.13 начинается с двух объявлений `using`:

```
using debts::Debt;        // делает доступным определение структуры Debt
using debts::showDebt;    // делает доступной функцию showDebt
```

Обратите внимание, что в объявлениях `using` присутствуют только имена. Так, во втором примере отсутствует описание типа возвращаемого значения и сигнатуры функции `showDebt`, а приводится лишь ее имя. (Таким образом, если функция была перегружена, одно объявление `using` обеспечит импортирование всех ее версий.) Кроме того, хотя `Debt` и `showDebt()` используют тип `Person`, нет необходимости им-

портировать какое-либо из имен Person, т.к. в пространстве имен debt уже содержится директива using, включающая пространство имен pers.

Функция other() использует менее желательный способ импорта всего пространства имен с помощью директивы using:

```
using namespace debts; // делает доступными для other() все имена из debts и pers
```

Поскольку директива using в debts импортирует пространство имен pers, функция other() может использовать тип данных Person и функцию showPerson().

Наконец, в функции another() применяется объявление using и операция разрешения контекста для доступа к отдельным именам:

```
using pers::Person;
pers::showPerson(collector);
```

Ниже показан результат выполнения программы, представленной в листингах 9.11, 9.12 и 9.13:

```
Goatsniff, Benny: $120
Glister, Doodles
Enter first name: Arabella
Enter last name: Binx
Enter debt: 100
Enter first name: Cleve
Enter last name: Delaproux
Enter debt: 120
Enter first name: Eddie
Enter last name: Fiotox
Enter debt: 200
Binx, Arabella: $100
Delaproux, Cleve: $120
Fiotox, Eddie: $200
Total debt: $420
Rightshift, Milo
```

Пространства имен и будущее

По мере освоения программистами пространств имен будет вырабатываться новый стиль программирования. Ниже представлены некоторые актуальные на данный момент рекомендации.

- Используйте переменные в именованных пространствах имен вместо внешних глобальных переменных.
- Используйте переменные в неименованных пространствах имен вместо статических глобальных переменных.
- Если вы разработали библиотеку функций или классов, поместите ее в пространство имен. Современный язык C++ уже требует помещения стандартных библиотечных функций в пространство имен std. Это же распространяется и на функции, унаследованные из C. Например, заголовочный файл math.c, который совместим с языком C, не использует пространства имен, но заголовочный файл cmath языка C++ предусматривает помещение различных математических функций в пространство имен std.
- Используйте директиву using только в качестве временного средства адаптации устаревшего кода к использованию пространств имен.

- Не применяйте директивы `using` в заголовочных файлах. Прежде всего, этот прием скрывает имена, которые сделаны доступными. Кроме того, порядок следования заголовочных файлов может влиять на поведение программы. Если вы используете директиву `using`, помещайте ее после всех директив препроцессора `#include`.
- Для импорта имен отдавайте предпочтение операции разрешения контекста или объявлению `using`.
- Для объявлений `using` отдавайте предпочтение локальной, а не глобальной области видимости.

Имейте в виду, что главная цель использования пространств имен состоит в упрощении управлением крупными проектами. Для простой программы, состоящей из одного файла, применение директивы `using` тоже не будет совершенно излишним.

Как уже упоминалось ранее, изменения в именах заголовочных файлов отражают изменения, связанные с пространствами имен. Заголовочные файлы старого стиля, такие как `iostream.h`, не работают с пространствами имен, а новый заголовочный файл `iostream` предусматривает использование пространства имен `std`.

Резюме

Особенности языка C++ благоприятствуют разработке программ, расположенных во множестве файлов. Эффективная стратегия организации программ состоит в применении заголовочного файла для определения пользовательских типов данных и прототипов функций, управляющих этими данными. Целесообразно помещать определения функций в отдельный файл исходного кода. Заголовочный файл и файл исходного кода совместно определяют и реализуют определяемый пользователем тип данных, а также средства для работы с ним. Функция `main()` и другие функции, использующие функции определенного типа, могут быть помещены в третий файл.

Схемы хранения C++ определяют, сколько переменные находятся в памяти (продолжительность хранения), а также какие части программы имеют к ним доступ (область видимости и связывание). Автоматическими называются переменные, которые определены внутри блока, такого как тело функции или блок внутри него. Они существуют и доступны только тогда, когда программа выполняет операторы блока, содержащего их определения. Автоматические переменные могут быть объявлены с помощью спецификатора класса хранения `register` или вообще без спецификатора; в последнем случае они становятся автоматическими по умолчанию. Спецификатор `register` служил подсказкой компилятору о том, что переменная интенсивно используется, но в C++11 он объявлен устаревшим.

Статические переменные существуют на протяжении всего периода выполнения программы. Переменная, определенная за пределами всех функций, известна всем функциям в данном файле, которые следуют за ее определением (область видимости файла), и доступна другим файлам программы (внешнее связывание). Для использования такой переменной в другом файле она должна быть объявлена с ключевым словом `extern`. Если переменная используется в нескольких файлах, один из файлов должен содержать ее определяющее объявление (ключевое слово `extern` использовать не обязательно, но его можно указать в комбинации с инициализацией), а остальные — ссылочные объявления (ключевое слово `extern` используется, но без инициализации). Переменная, объявленная вне функций, но снабженная ключевым словом `static`, обладает областью видимости файла, но не доступна другим файлам (внутреннее связывание).

Переменная, определенная внутри блока, но сопровождаемая ключевым словом static, является локальной по отношению к этому блоку (локальная область видимости без связывания), однако сохраняет свое значение в течение всего времени выполнения программы.

По умолчанию функции C++ имеют внешнее связывание, поэтому они могут разделяться между файлами. Однако функции с ключевым словом static имеют внутреннее связывание; их использование ограничено файлом, содержащим их определения.

Динамическое выделение и освобождение памяти с помощью new и delete использует для данных свободное хранилище, или кучу. Память становится доступной для использования при вызове new и освобождается при вызове delete. Для отслеживания адресов в этой памяти применяются указатели.

Пространства имен позволяют определять именованные области памяти, в которых можно объявлять идентификаторы. Они предназначены для снижения вероятности конфликта имен, что особенно важно в крупных программах, использующих код от различных поставщиков. Для обеспечения доступа к идентификаторам пространств имен может применяться операция разрешения контекста, объявление using или директива using.

Вопросы для самоконтроля

1. Какой схемой хранения вы воспользуетесь в следующих ситуациях?

   а. homer — это формальный аргумент (параметр) функции.

   б. Переменная secret должна совместно использоваться в двух файлах.

   в. Переменная topsecret должна быть доступна функциям одного файла, но скрыта от других файлов.

   г. Переменная beencalled фиксирует количество вызовов функции, которая ее содержит.

2. Опишите различия между объявлением using и директивой using.

3. Перепишите следующий код таким образом, чтобы в нем не использовалось ни объявление, ни директива using.

```
#include <iostream>
using namespace std;
int main()
{
    double x;
    cout << "Enter value: ";
    while (! (cin >> x) )
    {
        cout << "Bad input. Please enter a number: ";  // неверный ввод
        cin.clear();
        while (cin.get() != '\n')
            continue;
    }
    cout << "Value = " << x << endl;
    return 0;
}
```

4. Перепишите следующий код таким образом, чтобы в нем использовались объявления using вместо директивы using.

```
#include <iostream>
```

```
using namespace std;
int main()
{
   double x;
   cout << "Enter value: ";
   while (! (cin >> x) )
   {
      cout << "Bad input. Please enter a number: ";  // неверный ввод
      cin.clear();
      while (cin.get() != '\n')
         continue;
   }
   cout << "Value = " << x << endl;
   return 0;
}
```

5. Предположим, что функция `average(3,6)` должна возвращать значение `int`, которое является средним арифметическим двух аргументов типа `int`, когда она вызывается в одном файле, и значение `double`, которое является средним арифметическим от двух аргументов типа `int`, когда вызывается в другом файле одной и той же программы. Как это можно реализовать?

6. Какие данные будет выводить следующая программа, состоящая из двух файлов?

```
// file1.cpp
#include <iostream>
using namespace std;
void other();
void another();
int x = 10;
int y;
int main()
{
   cout << x << endl;
   {
      int x = 4;
      cout << x << endl;
      cout << y << endl;
   }
   other();
   another();
   return 0;
}
void other()
{
   int y=1;
   cout << "Other: " << x << ", " << y << end;
}
```

```
// file2.cpp
#include <iostream>
using namespace std;
extern int x;
namespace
{
   int y = -4;
}
```

```
void another()
{
    cout << "another(): " << x << ", " << y << endl;
}
```

7. Что будет выводить следующая программа?

```
#include <iostream>
using namespace std;
void other();
namespace n1
{
    int x = 1;
}
namespace n2
{
    int x = 2;
}
int main()
{
    using namespace n1;
    cout << x << endl;
    {
        int x = 4;
        cout << x <<  ", " << n1::x << ", " << n2::x << endl;
    }
    using n2::x;
    cout << x << endl;
    other();
    return 0;
}
void other()
{
    using namespace n2;
    cout << x << endl;
    {
        int x = 4;
        cout << x << ", " << n1::x << ", " << n2::x << endl;
    }
    using n2::x;
    cout << x << endl;
}
```

Упражнения по программированию

1. Имеется следующий заголовочный файл:

```
// golf.h -- для pe9-1.cpp
const int Len = 40;
struct golf
{
    char fullname[Len];
    int handicap;
};
```

```
// Неинтерактивная версия:
// функция присваивает структуре типа golf имя игрока и его гандикап (фору),
// используя передаваемые ей аргументы
void setgolf(golf & g, const char * name, int hc);

// Интерактивная версия:
// функция предлагает пользователю ввести имя и гандикап,
// присваивает элементам структуры g введенные значения;
// возвращает 1, если введено имя, и 0, если введена пустая строка
int setgolf(golf & g);

// Функция устанавливает новое значение гандикапа
void handicap(golf & g, int hc);

// Функция отображает содержимое структуры типа golf
void showgolf(const golf & g);
```

Обратите внимание, что функция `setgolf()` перегружена. Вызов первой версии функции имеет следующий вид:

```
golf ann
setgolf(ann, "Ann Birdfree", 24);
```

Функция предоставляет информацию, которая содержится в структуре `ann`. Вызов второй версии функции имеет следующий вид:

```
golf andy
setgolf(andy);
```

Функция предлагает пользователю ввести имя и гандикап, а затем сохраняет эти данные в структуре `andy`. Эта функция могла бы (но не обязательно) внутренне использовать первую версию.

Постройте многофайловую программу на основе этого заголовочного файла. Один файл по имени `golf.cpp` должен содержать определения функций, которые соответствуют прототипам заголовочного файла. Второй файл должен содержать функцию `main()` и обеспечивать реализацию всех средств прототипированных функций. Например, цикл должен запрашивать ввод массива структур типа `golf` и прекращать ввод после заполнения массива, либо когда вместо имени игрока в гольф пользователь вводит пустую строку. Чтобы получить доступ к структурам типа `golf`, функция `main()` должна использовать только прототипированные функции.

2. Модифицируйте код в листинге 9.9, заменив символьный массив объектом `string`. Программа больше не должна проверять, умещается ли вводимая строка, и для проверки ввода пустой строки может сравнивать вводимую строку со значением `""`.

3. Начните со следующего объявления структуры:

```
struct chaff
{
    char dross[20];
    int slag;
};
```

Напишите программу, которая использует операцию `new` с размещением, чтобы поместить массив из двух таких структур в буфер. Затем программа присваивает значения членам структуры (не забудьте воспользоваться функцией `strcpy()` для массива `char`) и отображает ее содержимое с помощью цикла. Вариант 1

предусматривает применение в качестве буфера памяти статического массива, как было показано в листинге 9.10. Вариант 2 состоит в использовании обычной операции new для выделения памяти под буфер.

4. Напишите программу, включающую три файла и использующую следующее пространство имен:

```
namespace SALES
{
   const int QUARTERS = 4;
   struct Sales
   {
      double sales[QUARTERS];
      double average;
      double max;
      double min;
   }
   // Копирует меньшее значение из 4 или n элементов из массива
   // ar в член sales структуры s, вычисляет и сохраняет
   // среднее арифметическое, максимальное и минимальное
   // значения введенных чисел;
   // оставшиеся элементы sales, если таковые есть, устанавливаются в 0
   void setSales(Sales & s, const double ar[], int n);

   // Интерактивно подсчитывает продажи за 4 квартала,
   // сохраняет их в члене sales структуры s, вычисляет и
   // сохраняет среднее арифметическое, а также максимальное
   // и минимальное значения введенных чисел
   void setSales(Sales & s);
   // Отображает всю информацию из структуры s
   void showSales(const Sales & s);
}
```

Первый файл должен быть заголовочным и содержать пространство имен. Второй файл должен содержать исходный код и расширять пространство имен, предоставляя определения трех прототипированных функций. В третьем файле должны объявляться два объекта Sales. Он должен использовать интерактивную версию функции setSales() для предоставления значений первой структуре и неинтерактивную версию той же функции для предоставления значений второй структуре. Он также должен отображать содержимое обеих структур с помощью функции showSales().

10

Объекты и классы

В ЭТОЙ ГЛАВЕ...

- Процедурное и объектно-ориентированное программирование
- Концепция классов
- Определение и реализация класса
- Открытый и закрытый доступ к классу
- Данные-члены класса
- Методы класса (также называемые функциями-членами класса)
- Создание и использование объектов класса
- Конструкторы и деструкторы класса
- Функции-члены `const`
- Указатель `this`
- Создание массивов объектов
- Область видимости класса
- Абстрактные типы данных

Объектно-ориентированное программирование (ООП) – это особый концептуальный подход к проектированию программ, и C++ расширяет язык C средствами, облегчающими применение такого подхода. Ниже перечислены наиболее важные характеристики ООП:

- абстракция;
- инкапсуляция и сокрытие данных;
- полиморфизм;
- наследование;
- повторное использование кода.

Класс – это единственное наиболее важное расширение C++, предназначенное для реализации этих средств и связывающее их между собой. Настоящая глава начинается с объяснения концепции классов. Здесь рассмотрены абстракция, инкапсуляция и сокрытие данных, а также показано, как эти средства реализуются в классах. В главе рассказывается об определении класса, о предоставлении класса с открытым и закрытым разделами, а также о создании функций-членов, которые работают с данными класса. Кроме того, вы ознакомитесь с конструкторами и деструкторами, которые представляют собой специальные функции-члены, предназначенные для создания и уничтожения объектов, относящихся к классу. И, наконец, вы узнаете, что такое указатель `this` – важный компонент для программирования некоторых классов. В последующих главах обсуждение будет продолжено описанием перегрузки операций (другая разновидность полиморфизма) и наследования – фундамента для повторного использования кода.

Процедурное и объектно-ориентированное программирование

Несмотря на то что настоящая книга в основном посвящена вопросам объектно-ориентированного программирования, стоит также более пристально взглянуть на стандартный процедурный подход, присущий таким языкам, как C, Pascal и BASIC. Давайте рассмотрим пример, демонстрирующий разницу между ООП и процедурным программированием.

Предположим, что вас, как нового члена софтбольной команды "Гиганты Жанра", попросили вести статистику игр команды. Естественно, вы используете для этого компьютер. Если вы – сторонник процедурного программирования, то, скорее всего, будете думать примерно так, как описано ниже.

> Итак, посмотрим... Я хочу вводить имя, количество подач, количество попаданий, средний уровень успешных подач (для тех, что не следит за бейсболом или софтболом: средний уровень успешных подач – это количество попаданий, деленное на официальное количество подач, выполненных игроком; подачи прекращаются, когда игрок достигает базы либо выбивает мяч за пределы поля, но некоторые события, такие как получение обводки, не засчитываются в официальное количество подач), а также другую существенную статистику по каждому игроку. Минутку, но ведь предполагается, что компьютер должен мне облегчать жизнь, поэтому я хочу, чтобы он вычислял автоматически кое-что из этого, например, средний уровень успешных подач. Кроме того, я хочу, чтобы программа выдавала отчет по результатам. Как это организовать? Думаю, это следует делать с помощью функций. Да, я сделаю, чтобы `main()` вызывала функцию для получения ввода, затем другую функцию – для вычислений, а потом третью – для

вывода результатов. Гм, а что случится, когда я получу данные о другой игре? Я не хочу начинать все сначала! Ладно, я могу добавить функцию для обновления статистики. Ну и ну, возможно, мне понадобится меню в `main()`, чтобы выбирать между вводом, вычислением, обновлением и выводом данных. Гм, а как представить данные? Можно использовать массив строк для хранения имен игроков, другой массив — для хранения числа подач каждого игрока и еще один массив — для хранения числа попаданий и т.д. Нет, это глупость! Я могу спроектировать структуру, чтобы хранить всю информацию о каждом игроке и затем использовать массив таких структур для представления всей команды.

Короче говоря, при процедурном подходе вы сначала концентрируетесь на процедурах, которым должны следовать, а только потом думаете о том, как представить данные. (Поскольку вам не нужно держать программу работающей в течение всего игрового сезона, вероятно, придется сохранять данные в файле и читать их оттуда.)

Теперь посмотрим, как изменится ваш взгляд, когда вы наденете шляпу ООП (сшитую в симпатичном полиморфном стиле). Вы начнете думать о данных. Более того, вы будете думать о данных не только в терминах их представления, но и в терминах их использования.

Итак, посмотрим, что я должен отслеживать? Игроков, конечно. Поэтому мне нужен объект, который представляет игрока в целом, а не только его уровень успешных подач или их общее количество. Да, это будет основная единица данных — объект, представляющий имя и статистику игрока. Мне понадобятся некоторые методы для работы с этим объектом. Гм, думаю, понадобится метод для ввода базовой информации в объект. Некоторые данные должен вычислять компьютер, например, средний уровень успешных подач. Я могу добавить метод для реализации вычислений. И программа должна выполнять вычисления автоматически, чтобы пользователю не нужно было помнить о том, что он должен просить ее об этом. Кроме того, понадобятся методы для обновления и отображения информации. То есть у пользователя должно быть три способа взаимодействия с данными: инициализация, обновление и вывод отчетов. Это и будет пользовательский интерфейс.

Короче говоря, при объектно-ориентированном подходе вы концентрируетесь на объекте, как его представляет пользователь, думая о данных, которые нужны для описания объекта, и операциях, описывающих взаимодействие пользователя с данными. После разработки описания интерфейса вы перейдете к выработке решений о том, как реализовать этот интерфейс и как организовать хранение данных. И, наконец, вы соберете все это вместе в программу, соответствующую новому проекту.

Абстракции и классы

Жизнь полна сложностей, и единственный способ справится со сложностью — это ограничиться упрощенными абстракциями. Человек состоит из более 10^{48} атомов. Некоторые утверждают, что сознание представляет собой коллекцию множества полуавтономных агентов. Но гораздо проще думать о себе, как о едином целом. В компьютерных вычислениях абстракция — это ключевой шаг в представлении информации в терминах ее интерфейса с пользователем. То есть вы абстрагируете основные операционные характеристики проблемы и выражаете решение в этих терминах. В примере с софтбольной статистикой интерфейс описывает, как пользователь инициирует, обновляет и отображает данные. От абстракций легко перейти к определяемым пользователем типам, которые в C++ представлены классами, реализующими абстрактный интерфейс.

Что такое тип?

Давайте подумаем немного о том, что собой представляет тип. Например, кто такой зануда? Если следовать популярному стереотипу, его можно представить в визуальных образах: толстый, в запотевших очках, карман, полный ручек, и т.п. После некоторых размышлений вы можете прийти к заключению, что зануду лучше описать с помощью присущего ему поведения — например, его реакцией в неловких ситуациях. Вы находитесь в похожем положении, если вы не имеете в виду искусственные аналогии, с процедурным языком, таким как C. Прежде всего, вы думаете о типе данных в терминах того, как он выглядит — каким образом хранится в памяти. Например, тип `char` занимает 1 байт памяти, а `double` — обычно 8 байт. Но если немного подумать, то вы придете к заключению, что тип данных также определен в терминах операций, которые допустимо выполнять с ним. Например, к типу `int` можно применять все арифметические операции. Целые числа можно складывать, вычитать, умножать, делить. Можно также применять операцию взятия модуля (`%`).

С другой стороны, рассмотрим указатели. Указатель может требовать для своего хранения столько же памяти, сколько и тип `int`. Он даже может иметь внутреннее представление в виде целого числа. Но указатель не позволяет выполнять над собой те же операции, что и целое. Например, перемножить два указателя не удастся. Эта концепция не имеет смысла, поэтому в C++ она не реализована. Таким образом, когда вы объявляете переменную как `int` или указатель на `float`, то не просто выделяете память для нее, но также устанавливаете, какие операции допустимы с этой переменной. Короче говоря, спецификация базового типа выполняет три вещи.

- Определяет, сколько памяти нужно объекту.
- Определяет, как интерпретируются биты памяти. (Типы `long` и `float` могут занимать одинаковое количество бит памяти, но транслируются в числовые значения по-разному.)
- Определяет, какие операции, или методы, могут быть применены с использованием этого объекта данных.

Для встроенных типов информация об операциях встроена в компилятор. Но когда вы определяете пользовательский тип в C++, то должны предоставить эту информацию самостоятельно. В качестве вознаграждения за эту дополнительную работу вы получаете мощь и гибкость новых типов данных, соответствующих требованиям реального мира.

Классы в C++

Класс — это двигатель C++, предназначенный для трансляции абстракций в пользовательские типы. Он комбинирует представление данных и методов для манипулирования этими данными в пределах одного аккуратного пакета. Давайте взглянем на класс, представляющий акционерный капитал.

Вначале нужно немного подумать о том, как представлять акции. Вы можете взять в качестве базовой единицы один пакет акций и определить класс, представляющий этот пакет. Однако это потребует создания 100 объектов для представления 100 пакетов, что явно не практично. Вместо этого в качестве базовой единицы можно представить персональную долю владельца в определенном пакете. Количество акций, находящихся во владении, будут частью представления данных.

Реалистичный подход должен позволять поддерживать хранение информации о таких вещах, как начальная стоимость и дата покупки; это необходимо для налогообложения. Кроме того, он должен предусматривать обработку таких событий, как раз-

деление пакетов. Это может показаться довольно амбициозным для первой попытки определения класса, поэтому можете ограничиться неким идеализированным, упрощенным взглядом на предмет. В частности, реализуемые операции можно ограничить следующим перечнем:

- приобретение пакета акций компании;
- приобретение дополнительных акций в имеющийся пакет;
- продажа пакета;
- обновление объема доли в пакете акций;
- отображения информации о пакетах, находящихся во владении.

Этот список можно использовать для определения открытого интерфейса класса (и при необходимости позже добавлять дополнительные средства). Для поддержки этого интерфейса необходимо сохранять некоторую информацию. И снова можно воспользоваться упрощенным подходом. Например, не нужно беспокоиться о принятой в США практике оперировать пакетами акций в объемах, кратных 8 долларам. (Видимо, на Нью-Йоркской фондовой бирже заметили это упрощение в предыдущем издании книги, потому что решили перейти к системе, которая описана здесь.) Ниже приведен список сведений для сохранения:

- название компании;
- количество акций, находящихся во владении;
- объем каждой доли;
- общий объем всех долей.

Далее можно определить класс. Обычно спецификация класса состоит из двух частей.

- *Объявление класса*, описывающее компоненты данных в терминах членов данных, а также открытый интерфейс в терминах функций-членов, называемых *методами*.
- *Определения методов класса*, которые описывают, как реализованы определенные функции-члены.

Грубо говоря, объявление класса предоставляет общий обзор класса, в то время как определения методов снабжают необходимыми деталями.

Что такое интерфейс?

Интерфейс — это совместно используемая часть, предназначенная для взаимодействия двух систем, например, между компьютером и принтером или между пользователем и компьютерной программой. Например, пользователем можете быть вы, а программой — текстовый процессор. Когда вы работаете с текстовым процессором, то не переносите слова напрямую из своего сознания в память компьютера. Вместо этого вы взаимодействуете с интерфейсом, предложенным программой. Вы нажимаете клавишу, и компьютер отображает символ на экране. Вы перемещаете мышь, и компьютер перемещает курсор на экран Вы случайно щелкаете кнопкой мышки, и абзац, в котором вы печатаете, искажается. Интерфейс программы управляет преобразованием ваших намерений в специфическую информацию, сохраняемую в компьютере. В отношении классов мы говорим об открытом интерфейсе. В этом случае потребителем его является программа, использующая класс, система взаимодействия состоит из объектов класса, а интерфейс состоит из методов, предоставленных тем, кто написал этот класс. Интерфейс позволяет вам, как программисту, написать код, взаимодействующий с объектами класса, и таким образом, дает программе возможность взаимодействовать с объектами класса.

Например, чтобы определить количество символов в объекте `string`, вам не нужно открывать этот объект и смотреть что у него внутри. Вы просто используете метод `size()` класса, предоставленный его разработчиком. Таким образом, метод `size()` является частью открытого интерфейса между пользователем и объектом класса `string`. Аналогичным образом метод `getline()` является частью открытого интерфейса класса `istream`. Программа, использующая `cin`, не обращается напрямую к внутренностям объекта `cin` для чтения строки ввода; вместо этого всю работу выполняет `getline()`.

Если вам нужны более персонализированные отношения, то вместо того, чтобы думать о программе, использующей класс, как о внешнем пользователе, вы можете думать об авторе программы, использующей этот класс, как о внешнем пользователе. Но в любом случае для работы с классом необходимо знать его открытый интерфейс, и для написания класса потребуется создать такой интерфейс.

Разработка классов и программ, которые их используют, требует выполнения определенных шагов. Вместо того чтобы взять на вооружение их целиком, давайте разделим процесс разработки на небольшие стадии. Обычно программисты на C++ помещают интерфейс, имеющий форму определения класса, в заголовочный файл, а реализацию в форме кода для методов класса — в файл исходного кода. Так что давайте не отступать от обычной практики. В листинге 10.1 представлена первая стадия, пробное объявление класса под именем `Stock`. В этом файле применяется `#ifndef` и т.п. (см. главу 9) для защиты против многократного включения файла.

Чтобы упростить идентификацию классов, в настоящей книге используется общий, однако не универсальный метод — соглашение о написании имен классов с заглавной буквы. Обратите внимание, что код в листинге 10.1 выглядит как объявление структуры с несколькими небольшими дополнениями, такими как функции-члены, а также разделы `public` и `private`. Вскоре вы улучшим это объявление (поэтому не используйте его в качестве модели), но вначале давайте посмотрим, как оно работает.

Листинг 10.1. `stock00.h`

```
// stock00.h — интерфейс класса Stock
// версия 00
#ifndef STOCK00_H_
#define STOCK00_H_

#include <string>

class Stock  // объявление класса
{
private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    void acquire(const std::string & co, long n, double pr);
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show();
}; // Обратите внимание на точку с запятой в конце.

#endif
```

На детали реализации класса мы взглянем позднее, а сейчас проанализируем наиболее общие средства. Первым делом, ключевое слово `class` в C++ идентифицирует код в листинге 10.1 как определение класса. (В таком контексте ключевые слова `class` и `typename` не являются синонимами, как это было в параметрах шаблона; `typename` здесь использовать нельзя.) Данный синтаксис идентифицирует `Stock` в качестве имени типа для нового класса. Это позволяет объявлять переменные, которые называются *объектами* или *экземплярами* типа `Stock`. Каждый индивидуальный объект этого типа представляет отдельный пакет акций, находящийся во владении. Например, следующее объявление создает два объекта с именами `sally` и `solly`:

```
Stock                                                    sally;
Stock                                                    solly;
```

Объект `sally`, например, мог бы представлять пакет акций определенной компании, принадлежащий Салли.

Далее обратите внимание, что информация, которую вы решили сохранять, появляется в форме данных-членов класса, таких как `company` и `shares`. Член `company` объекта `sally`, например, хранит название компании, член `share` содержит количество долей общего пакета акций компании, которыми владеет Салли, член `share_val` соответствует объему каждой доли, а член `total_val` — общему объему всех долей. Аналогично необходимые операции представлены в виде функций-членов (или методов), таких как `sell()` и `update()`. Функция-член может быть определена на месте, как, например, `set_tot()`, либо подобно остальным функции-члены в этом классе представлена с помощью прототипа. Полные определения остальных функций-членов появятся позже, в файле реализации, но прототипов уже достаточно для описания их интерфейса. Связывание данных и методов в единое целое — наиболее замечательное свойство класса. При таком проектном решении создание объекта типа `Stock` автоматически устанавливает правила, регулирующие его использованием.

Вы уже видели, что классы `istream` и `ostream` имеют функции-члены вроде `get()` и `getline()`. Прототипы функций в определении класса `Stock` демонстрируют установку функций-членов. Заголовочный файл `iostream`, например, содержит прототип `getline()` в объявлении класса `iostream`.

Управление доступом

Новыми также являются ключевые слова `private` и `public`. Эти метки позволяют *управлять доступом* к членам класса. Любая программа, которая использует объект определенного класса, может иметь непосредственный доступ к членам из раздела `public`. Доступ к членам объекта из раздела `private` программа может получить *только* через открытые функции-члены из раздела `public` (или же, как будет показано в главе 11, через дружественные функции). Например, единственный способ изменить переменную `shares` класса `Stock` — это воспользоваться одной из функций-членов класса `Stock`. Таким образом, открытые функции-члены действуют в качестве посредников между программой и закрытыми членами объекта; они предоставляют интерфейс между объектом и программой. Эта изоляция данных от прямого доступа со стороны программы называется *сокрытием данных*. (В C++ имеется третье ключевое слово для управления доступом — `protected`, которое объясняется при обсуждении наследовании в главе 13.) Все сказанное иллюстрируется на рис. 10.1. В то время как сокрытие данных может быть недобросовестным действием, когда мы говорим в общепринятом контексте о доступе к информации инвестиционных фондов, это является хорошей практикой в компьютерных вычислениях, поскольку предохраняет целостность данных.

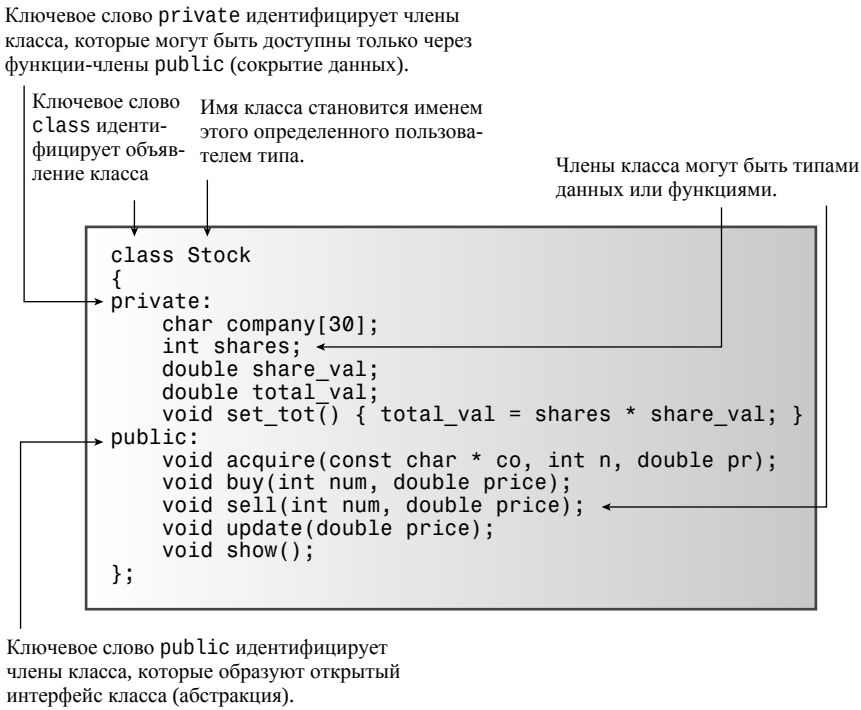


Рис. 10.1. *Класс* `Stock`

Проектное решение класса пытается отделить открытый интерфейс от специфики реализации. Открытый интерфейс представляет абстрактный компонент проектного решения. Собрание деталей реализации в одном месте и отделение их от абстракции называется *инкапсуляцией*. *Сокрытие данных* (помещение данных в раздел `private` класса) является примером инкапсуляции, и поэтому оно скрывает функциональные детали реализации в разделе `private`, как это сделано в классе `Stock` с функцией `set_tot()`. Другим примером инкапсуляции может служить обычная практика помещения определений функций класса в файл, отдельный от объявления класса.

Объектно-ориентированное программирование и C++

Объектно-ориентированное программирование (ООП) — это стиль программирования, который в той или иной степени можно применять в любом языке. Безусловно, вы можете реализовать многие идеи ООП в обычной программе на языке C. Например, в главе 9 представлен пример (см. листинги 9.1, 9.2, 9.3), в котором заголовочный файл содержит прототип структуры вместе с прототипами функций, предназначенных для манипулирования этой структурой. Функция `main()` просто определяет переменные этого типа и использует ассоциированные функции для управления этими переменными; `main()` не имеет непосредственного доступа к членам структур. По сути, в этом примере объявляется абстрактный тип, который помещает формат хранения и прототипы функций в заголовочный файл, скрывая реальное представление данных от `main()`.

Язык C++ включает средства, специально предназначенные для реализации подхода ООП, что позволяет продвинуться в этом направлении на несколько шагов дальше, чем в языке C. Во-первых, размещение прототипов функций в едином объявлении класса вместо того, чтобы держать их раздельно, унифицирует описание за счет размещения его в одном месте.

Во-вторых, объявление данных с закрытым доступом разрешает доступ к ним только для авторизованных функций. Если в примере на C функция `main()` имеет непосредственный доступ к членам структуры, то это противоречит духу ООП, но не нарушает никаких правил языка C. Однако попытка прямого доступа, скажем, к члену `shares` объекта `Stock` приводит к нарушению правила языка C++, и компилятор перехватит это.

Следует отметить, что сокрытие данных не только предотвращает прямой доступ к данным, но также избавляет вас (в роли пользователя этого класса) от необходимости знать то, как представлены данные. Например, член `show()` отображает, помимо прочего, общую сумму пакета акций, находящегося во владении. Это значение может быть сохранено в виде части объекта, как это делает код в листинге 10.1, либо при необходимости может быть вычислено. С точки зрения пользователя класса нет разницы, какой подход применяется. Необходимо знать только то, что делают различные функции-члены — т.е. какие аргументы они принимают, и какие типы значений возвращают. Принцип состоит в том, чтобы отделить детали реализации от проектного решения интерфейса. Если позже вы найдете лучший способ реализации представления данных или деталей внутреннего устройства функций-членов, то сможете изменить их без изменения программного интерфейса, что значительно облегчает поддержку и сопровождение программ.

Управление доступом к членам: `public` или `private`?

Объявлять члены класса — будь они элементами данных или функциями-членами — можно как в открытом (`public`), так и в закрытом (`private`) разделе класса. Но поскольку одним из главных принципов ООП является сокрытие данных, то единицы данных обычно размещаются в разделе `private`. Функции-члены, которые образуют интерфейс класса, размещаются в разделе `public`; в противном случае вызвать эти функции из программы не удастся. Как показывает объявление класса `Stock`, вы все же можете поместить функции-члены в раздел `private`. Вызвать такие функции из программы непосредственно не получится, но их могут использовать открытые методы. Как правило, закрытые функции-члены применяются для управления деталями реализации, которые не формируют часть открытого интерфейса.

Использовать ключевое слово `private` в объявлении класса не обязательно, поскольку это спецификатор доступа к объектам класса по умолчанию:

```
class World
{
   float mass;            // по умолчанию private
   char name[20];         // по умолчанию private
public:
   void tellall(void);
   ...
}
```

Однако в этой книге метка `private` будет указываться явно, чтобы подчеркнуть концепцию сокрытия данных.

Классы и структуры

Описания классов выглядят очень похожими на объявления структур с дополнениями в виде функций-членов и меток видимости `private` и `public`. Фактически C++ расширяет на структуры те же самые свойства, которые есть у классов. Единственная разница состоит в том, что типом доступа по умолчанию у структур является `public`, в то время как у классов — `private`. Программисты на C++ обычно используют классы для реализации описаний классов, тогда как ограниченные структуры применяются для чистых объектов данных (которые часто называются *простыми старыми структурами данных* (plain-old data — POD)).

Реализация функций-членов класса

Мы по-прежнему обязаны определять вторую часть спецификации класса, т.е. предоставлять код для тех функций-членов, которые описаны с помощью прототипов в объявлении класса. Определения функций-членов очень похожи на определения обычных функций. Каждое из них имеет заголовок и тело. Определения функций-членов могут иметь тип возврата и аргументы. Но, кроме того, с ними связаны две специфических характеристики.

- При определении функции-члена для идентификации класса, которому принадлежит функция, используется операция разрешения контекста (`::`).
- Методы класса имеют доступ к `private`-компонентам класса.

Давайте рассмотрим все это подробнее.

В заголовке функции-члена для идентификации класса, которому она принадлежит, применяется операция разрешения контекста (`::`). Например, заголовок для функции-члена `update()` выглядит следующим образом:

```
void Stock::update(double price)
```

Эта нотация означает, что вы определяете функцию `update()`, которая является членом класса `Stock`. Но это означает не только то, что функция `update()` является функцией-членом, но также и то, что такое же имя можно использовать для функций-членов другого класса. Например, функция `update()` для класса `Button` будет иметь следующий заголовок:

```
void Button::update(double price)
```

Таким образом, операция разрешения контекста идентифицирует класс, к которому данный метод относится. Говорят, что идентификатор `update()` имеет область видимости класса. Другие функции-члены класса `Stock` могут при необходимости использовать метод `update()` без операции разрешения контекста. Это связано с тем, что они принадлежат одному классу, и, следовательно, имеют общую область видимости. Использование `update()` за пределами объявления класса и определений методов, однако, требует соблюдения специальных мер, которые мы рассмотрим далее.

Единственный способ однозначного разрешения имен методов — использовать полное имя, включающее имя класса. `Stock::update()` называется *уточненным именем* функции. Простое имя `update()`, с другой стороны, является сокращением (*не уточненным именем*) полного имени и может применяться только в контексте класса.

Следующей специальной характеристикой методов является то, что метод может иметь доступ к закрытым членам класса. Например, метод `show()` может использовать код вроде следующего:

```
cout << "Company: " << company
     << " Shares: " << shares << endl
     << " Share Price: $" << share_val
     << " Total Worth: $" << total_val << endl;
```

Здесь `company`, `shares` и т.д. являются закрытыми данными-членами класса `Stock`. Если вы попытаетесь воспользоваться для доступа к этим данным-членам функцией, которая не является членом, то компилятор воспрепятствует этому. (Исключением являются дружественные функции, описанные в главе 11.)

Памятуя об этих двух обстоятельствах, методы класса можно реализовать, как показано в листинге 10.2. Эти определения методов могут быть помещены в отдельный файл либо в тот же файл, где находится объявление класса. Мы поместили их в от-

дельный файл реализации, поэтому в нем потребуется включить заголовочный файл `stock00.h`, чтобы компилятор имел доступ к определению класса. Для демонстрации возможности работы с пространствами имен в одних методах используется квалификатор `std::`, а в других — объявления `using`.

Листинг 10.2. `stocks00.cpp`

```
// stock00.cpp -- реализация класса Stock
// версия 00
#include <iostream>
#include "stock00.h"
void Stock::acquire(const std::string & co, long n, double pr)
{
    company = co;
    if (n < 0)
    {
        // Количество пакетов не может быть отрицательным; устанавливается в 0.
        std::cout << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}

void Stock::buy(long num, double price)
{
    if (num < 0)
    {
        //Количество приобретаемых пакетов не может быть отрицательным. Транзакция прервана.
        std::cout << "Number of shares purchased can't be negative. "
                  << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}

void Stock::sell(long num, double price)
{
    using std::cout;
    if (num < 0)
    {
        // Количество продаваемых пакетов не может быть отрицательным. Транзакция прервана.
        cout << "Number of shares sold can't be negative. "
             << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        // Нельзя продать больше того, чем находится во владении. Транзакция прервана.
        cout << "You can't sell more than you have! "
             << "Transaction is aborted.\n";
    }
```

```
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}
void Stock::update(double price)
{
    share_val = price;
    set_tot();
}
void Stock::show()
{
    // Вывод названия компании, количества пакетов, цены пакета и общей стоимости.
    std::cout << "Company: " << company
              << " Shares: " << shares << '\n'
              << " Share Price: $" << share_val
              << " Total Worth: $" << total_val << '\n';
}
```

Замечания о функциях-членах

Функция `acquire()` управляет первоначальной покупкой пакета акций заданной компании, в то время как `buy()` и `sell()` — дополнительной покупкой и продажей акций из существующего пакета. Методы `buy()` и `sell()` гарантируют, что количество купленных или проданных акций не будет отрицательным. Кроме того, если пользователь пытается продать больше акций, чем у него есть, функция `sell()` отменит транзакцию. Прием объявления данных закрытыми и ограничения доступа к открытым функциям предоставляет контроль над использованием данных. В данном случае это позволяет предпринять защитные меры против недопустимых транзакций.

Четыре из определенных функций-членов устанавливают или сбрасывают значение члена `total_val`. Вместо того чтобы повторять в коде вычисление этого значения четыре раза, каждая из открытых функций-членов вызывает функцию `set_tot()`. Поскольку эта функция представляет собой просто реализацию внутреннего кода, а не является частью открытого интерфейса, в классе она объявлена как закрытая функция-член. (То есть `set_tot()` представляет собой функцию-член, которая используется разработчиком класса, но не теми, кто пишет код, использующий класс.) Если вычисления, выполняемые функцией, оказываются сложными, это поможет также уменьшить общий объем исходного кода. Однако здесь главное в том, что за счет использования вызова этой функции вместо повторения кода вычислений обеспечивается гарантия того, что всегда будет применяться абсолютно идентичный алгоритм. Кроме того, если его понадобится изменить (хотя в данном конкретном случае такое маловероятно), это нужно будет сделать только в одном месте.

Встроенные методы

Любая функция с определением внутри объявления класса автоматически становится встроенной. Это значит, что `Stock::set_tot()` является встроенной функцией. Объявления класса часто используют встроенные функции для коротких функций-членов, и `set_tot()` — пример такого случая.

Если хотите, можете определить функцию-член вне объявления класса и, тем не менее, сделать ее встроенной. Чтобы это сделать, просто используйте квалификатор `inline` при определении функции в разделе реализации класса:

```
class Stock
{
private:
   ...
   void set_tot();            // определение оставлено отдельным
public:
   ...
};
inline void Stock::set_tot()  // использование inline в определении
{
   total_val = shares * share_val;
}
```

Специальные правила для встроенных функций требуют, чтобы они были определены в каждом файле, в котором используются. Самый простой способ гарантировать, что встроенные определения доступны всем файлам в многофайловой программе — поместить эти определения в тот же заголовочный файл, где объявлен класс. (Некоторые системы разработки снабжены интеллектуальными компоновщиками, которые разрешают размещать встроенные определения в отдельном файле реализации.)

Кстати, согласно *правилу перезаписи*, определение метода внутри объявления класса эквивалентно замене определения метода прототипом и последующей перезаписью определения в виде встроенной функции немедленно после объявления класса. То есть исходное определение `set_tot()` в листинге 10.1 эквивалентно только что показанному, где определение следует за объявлением класса.

Какой объект использует метод?

Мы подошли к одному из наиболее важных аспектов использования объектов: каким образом метод класса применяется к объекту. Код вроде показанного ниже использует член `shares` некоторого объекта:

```
shares += num;
```

Но какого объекта? Отличный вопрос! Чтобы ответить на него, давайте вначале посмотрим, как создается объект. Наиболее простой способ объявления переменных класса выглядит так:

```
Stock kate, joe;
```

Это создает два объекта класса `Stock`, один по имени `kate`, а второй — `joe`.

Теперь рассмотрим, как использовать функцию-член с одним из этих объектов. Ответ, как и случае структур и членов структур, состоит в применении операции членства:

```
kate.show();   // объект kate вызывает функцию-член
joe.show();    // объект joe вызывает функцию-член
```

Первый оператор вызывает `show()` как член объекта `kate`. Это значит, что метод интерпретирует `shares` как `kate.shares`, а `share_val` — как `kate.share_val`. Аналогично вызов `joe.show()` заставляет метод `show()` интерпретировать `shares` как `joe.shares`, а `share_val` — как `joe.share_val`, соответственно.

На заметку

Когда вы вызываете функцию-член, она использует данные-члены конкретного объекта, примененного для ее вызова.

Подобным же образом вызов `kate.sell()` запускает функцию `set_tot()`, как если бы это была `kate.set_tot()`, позволяя ей получать доступ к данным объекта `kate`.

Каждый вновь созданный вами объект содержит хранилище для собственных внутренних переменных-членов класса, однако все объекты одного класса разделяют общий набор методов, по одной копии каждого. Предположим, например, что `kate` и `joe` — это объекты класса `Stock`. В этом случае `kate.shares` занимает один фрагмент памяти, а `joe.shares` — другой. Но `kate.show()` и `joe.show()` представляют собой один и тот же метод, т.е. оба выполняют один и тот же блок кода, только применяют этот код к разным данным. Вызов функции-члена — это то, что в некоторых объектно-ориентированных языках называется *отправкой сообщения*. Таким образом, отправка сообщения двум разным объектам вызывает один и тот же метод, который применяется к двум разным объектам (рис. 10.2).

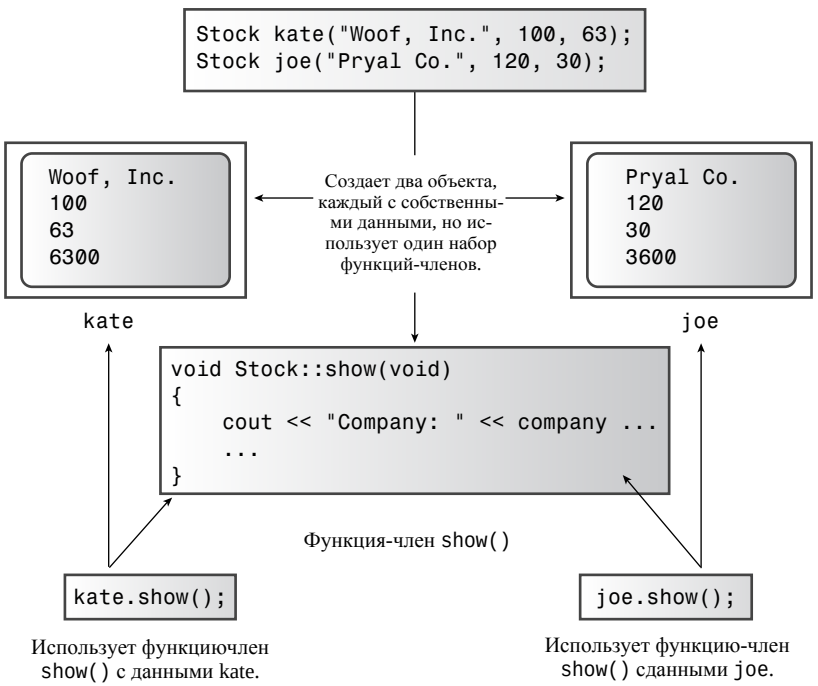


Рис. 10.2. Объекты, данные и функции-члены

Использование классов

В настоящей главе было показано, как определять класс и его методы. Следующий шаг состоит в разработке программы, которая будет создавать и использовать объекты класса. Целью языка C++ является сделать применение классов насколько возможно простым — подобно базовым встроенным типам вроде `int` и `char`. Создавать объект класса можно за счет объявления переменной этого класса либо использования операция `new` для размещения в памяти объекта этого класса. Объекты можно передавать в аргументах, возвращать их из функций, присваивать один объект другому.

Язык C++ предоставляет средства для инициализации объектов, для обучения cin и cout распознавать объекты и даже для выполнения автоматического приведения типов между объектами подобных классов. Пройдет некоторое время, прежде чем вы научитесь делать все эти вещи, но давайте начнем с наиболее простых свойств. Несомненно, вы уже видели, как объявлять объекты класса и вызывать функции-члены. В листинге 10.3 приведен код программы, которая использует файлы интерфейса и реализации. В коде создается объект типа Stock по имени fluffy_the_cat.

Программа проста, тем не менее, она проверяет все средства, которые вы встроите в класс. Для компиляции полной программы применяйте приемы, предназначенные для многофайловых программ, которые были описаны в главах 1 и 9. В частности, компилируйте ее с файлом stock00.cpp и обеспечьте наличие файла stock00.cpp в том же каталоге или папке.

Листинг 10.3. usestok0.cpp

```
// usestck0.cpp -- клиентская программа
// Компилируется вместе с stock00.cpp
#include <iostream>
#include "stock00.h"
int main()
{
    Stock fluffy_the_cat;
    fluffy_the_cat.acquire("NanoSmart", 20, 12.50);
    fluffy_the_cat.show();
    fluffy_the_cat.buy(15, 18.125);
    fluffy_the_cat.show();
    fluffy_the_cat.sell(400, 20.00);
    fluffy_the_cat.show();
    fluffy_the_cat.buy(300000,40.125);
    fluffy_the_cat.show();
    fluffy_the_cat.sell(300000,0.125);
    fluffy_the_cat.show();
    return 0;
}
```

Ниже показан вывод программы из листинга 10.3:

```
Company: NanoSmart  Shares: 20
  Share Price: $12.5  Total Worth: $250
Company: NanoSmart  Shares: 35
  Share Price: $18.125  Total Worth: $634.375
You can't sell more than you have! Transaction is aborted.
Company: NanoSmart  Shares: 35
  Share Price: $18.125  Total Worth: $634.375
Company: NanoSmart  Shares: 300035
  Share Price: $40.125  Total Worth: $1.20389e+007
Company: NanoSmart  Shares: 35
  Share Price: $0.125  Total Worth: $4.375
```

Обратите внимание, что main() — это просто механизм для тестирования класса Stock. Когда класс Stock заработает должным образом, его можно будет применять в качестве пользовательского типа в других программах. Важнейшим моментом для использования нового типа является понимание того, что делают функции-члены; вы не должны задумываться о деталях реализации. См. следующую врезку "Клиент-серверная модель".

Клиент–серверная модель

Программисты, соблюдающие принципы ООП, часто обсуждают проект программ в терминах клиент-серверной модели. Согласно этой концепции, *клиентом* является программа, которая использует класс. Объявление класса, включая его методы, образует *сервер*, который является ресурсом, доступным нуждающейся в нем программе. Клиент взаимодействует с сервером только через открытый (`public`) интерфейс. Это означает, что единственной ответственностью клиента и, как следствие — программиста, является знание интерфейса. Ответственностью сервера и, как следствие — его разработчика, является обеспечение того, чтобы его реализация надежно и точно соответствовала интерфейсу. Любые изменения, вносимые разработчиком сервера в класс, должны касаться деталей реализации, но не интерфейса. Это позволяет программистам разрабатывать клиент и сервер независимо друг от друга, без внесения в сервер таких изменений, которые нежелательным образом отобразятся на поведении клиента.

Изменение реализации

С выводом программы связан один момент, который может не устраивать — неподходящее форматирование чисел. Имеется возможность улучшить реализацию, не затрагивая интерфейс. Класс `ostream` содержит функции-члены, которые управляют форматированием. Не особо вдаваясь в детали, скажем, что с помощью метода `setf()` можно избавиться от экспоненциальной нотации, как это уже делалось в листинге 8.8:

```
std::cout.setf(std::ios_base::fixed, std::ios_base::floatfield);
```

Этот вызов устанавливает флаг, который заставляет объект `cout` использовать нотацию с фиксированной точкой. Подобным же образом следующий оператор заставляет `cout` выводить три десятичных знака после точки:

```
std::cout.precision(3);
```

Дополнительные сведения можно найти в главе 17.

Эти средства можно использовать в методе `show()` для управления форматированием, но следует учесть еще один момент. В случае изменения реализации метода внесенные модификации не должны влиять на другие части клиентской программы. Изменения в формате будут оставаться активными вплоть до следующих изменений, поэтому они могут повлиять на последующий вывод в клиентской программе. Следовательно, в `show()` должен быть предусмотрен возврат к состоянию форматирования, которое было до вызова этого метода. Это можно сделать, как и в листинге 8.8, с применением возвращаемых значений операторов установки формата:

```
std::streamsize prec =
   std::cout.precision(3);    // сохранение предыдущего значения точности
...
std::cout.precision(prec);    // восстановление предыдущего значения

// Сохранение исходных флагов
std::ios_base::fmtflags orig = std::cout.setf(std::ios_base::fixed);
...
// Восстановление сохраненных значений
std::cout.setf(orig, std::ios_base::floatfield);
```

Во-первых, вспомните, что `fmtflags` — это тип, определенный в классе `ios_base`, который находится в пространстве имен `std`, отсюда и такое довольно длинное имя типа для `orig`. Во-вторых, `orig` хранит все флаги, и оператор сброса использует эту информацию для восстановления установок в разделе `floatfield`, который включает флаги для нотации с фиксированной точкой и экспоненциальной нотации. В-третьих, давайте не

будем здесь сильно беспокоиться о деталях. Главный момент в том, что изменения ограничиваются файлом реализации и не влияют на программу, использующую этот класс. Итак, изменим определение метода в файле реализации следующим образом:

```
void Stock::show()
{
    using std::cout;
    using std::ios_base;
    // Установка формата в #.###
    ios_base::fmtflags orig =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    std::streamsize prec = cout.precision(3);
    cout << "Company: " << company
         << " Shares: " << shares << '\n';
    cout << " Share Price: $" << share_val;
    // Установка формата в #.##
    cout.precision(2);
    cout << " Total Worth: $" << total_val << '\n';
    // Восстановление исходного формата
    cout.setf(orig, ios_base::floatfield);
    cout.precision(prec);
}
```

После этой замены программу можно перекомпилировать. Теперь вывод будет выглядеть так:

```
Company: NanoSmart  Shares: 20
  Share Price: $12.500  Total Worth: $250.00
Company: NanoSmart  Shares: 35
  Share Price: $18.125  Total Worth: $634.38
You can't sell more than you have! Transaction is aborted.
Company: NanoSmart  Shares: 35
  Share Price: $18.125  Total Worth: $634.38
Company: NanoSmart  Shares: 300035
  Share Price: $40.125  Total Worth: $12038904.38
Company: NanoSmart  Shares: 35
  Share Price: $0.125  Total Worth: $4.38
```

Обзор ситуации на текущий момент

Первый шаг в проектировании класса заключается в предоставлении объявления класса. Объявление класса смоделировано на основе объявления структуры и может включать в себя данные-члены и функции-члены. Объявление имеет раздел `private`, и члены, объявленные в этом разделе, могут быть доступны только через функции-члены. Объявление также содержит раздел `public`, и объявленные в нем члены могут быть непосредственно доступны программе, использующей объекты класса. Как правило, данные-члены попадают в закрытый раздел, а функции-члены — в открытый, поэтому типичное объявление класса имеет следующую форму:

```
class имяКласса
{
private:
    объявления данных-членов
public:
    прототипы функций-членов
};
```

Содержимое открытого раздела включает абстрактную часть проектного решения — открытый интерфейс. Инкапсуляция данных в закрытом разделе защищает их целостность и называется сокрытием данных. Таким образом, использование классов — это способ, который предлагает C++ для облегчения реализации абстракций, сокрытия данных и инкапсуляции ООП.

Второй шаг в спецификации класса — это реализация функций-членов класса. Вместо прототипов в объявление можно включать полное определение функций, однако общепринятая практика состоит в том, чтобы определять функции отдельно, за исключением наиболее простых. В этом случае вам понадобится операция разрешения контекста для индикации того, к какому классу данная функция-член принадлежит. Например, предположим, что класс `Bozo` имеет функцию-член `Retort()`, которая возвращает указатель на тип `char`. Заголовок функции должен выглядеть примерно так:

```
char * Bozo::Retort()
```

Другими словами, `Retort()` — не только функция типа `char *`, это функция типа `char *`, принадлежащая классу `Bozo`. Полное, или уточненное, имя функции будет выглядеть как `Bozo::Retort()`. Имя `Retort()`, с другой стороны, является сокращением уточненного имени, и оно должно использоваться только в определенных случаях, таких как в коде методов класса.

Другой способ описания этой ситуации — это говорить о том, что `Retort` имеет область видимости класса, поэтому необходима операция разрешения контекста для уточнения имени, когда оно встречается вне объявления и вне методов класса.

Для создания объекта, который является частным примером класса, применяется имя класса, как если бы оно было именем типа:

```
Bozo bozetta;
```

Это работает потому, что класс является типом, определенным пользователем.

Функция-член класса, или метод, вызывается с использованием объекта класса. Это делается с помощью операции членства (точки):

```
cout << bozetta.Retort();
```

Код вызывает функцию-член `Retort()`, и всякий раз, когда код этой функции обращается к определенным данным-членам, используются значения членов объекта `bozetta`.

Конструкторы и деструкторы классов

Теперь нам нужно сделать с классом `Stock` нечто большее. Существует ряд определенных стандартных функций, называемых *конструкторами* и *деструкторами*, которыми обычно должен быть снабжен класс. Давайте поговорим о том, почему они необходимы и как их создавать.

Одна из целей C++ состоит в том, чтобы сделать использование объектов классов подобным применению стандартных типов. Однако код, который был приведен до сих пор в настоящей главе, не позволяет инициализировать объект `Stock` таким же способом, как это можно сделать с `int` или `struct`. То есть обычный синтаксис инициализации не применим к типу `Stock`:

```
int year = 2001;    // допустимая инициализация
struct thing
{
    char * pn;
    int m;
};
```

```
thing amabob = {"wodget", -23};                    // допустимая инициализация
Stock hot = {"Sukie's Autos, Inc.", 200, 50.25}; // ошибка компиляции
```

Причина, по которой нельзя таким способом инициализировать объект `Stock`, связана с тем, что к данным класса разрешен только закрытый доступ, а это означает, что единственный способ, с помощью которого программа может получить доступ к ним — через функции-члены. Следовательно, для успешной инициализации объекта понадобится придумать соответствующую функцию-член. (Чтобы инициализировать объект класса так, как показано выше, нужно объявить данные-члены как `public`, а не `private`, но в этом случае нарушается один из базовых принципов использования классов — сокрытие данных.)

В общем случае лучше, чтобы все объекты инициализировались при их создании. Например, рассмотрим следующий код:

```
Stock gift;
gift.buy(10, 24.75);
```

При существующей реализации класса `Stock` объект `gift` не имеет установленного значения для члена `company`. В проектном решении класса предполагается, что пользователь вызовет `acquire()` раньше любых других функций-членов, однако нет какого-либо средства, чтобы гарантировать это. Единственным способом обойти эту трудность является автоматическая инициализация объектов при их создании. Для этого в C++ предлагаются специальные функции-члены, называемые *конструкторами класса*, которые предназначены для создания новых объектов и присваивания значений их членам-данным. Если говорить точнее, то C++ регламентирует имя для таких функций-членов, а также синтаксис их вызова, тогда как ваша задача — написать определение этого метода. Имя метода конструктора совпадает с именем класса. Например, возможный конструктор для класса `Stock` — это функция-член `Stock()`. Прототип и заголовок конструктора обладают интересным свойством: несмотря на то, что конструкторы не имеют возвращаемого значения, они не объявляются с типом `void`. Фактически конструкторы не имеют объявленного типа.

Объявление и определение конструкторов

Теперь потребуется написать конструктор `Stock`. Поскольку объект `Stock` имеет три значения, которые ему нужно получить из внешнего мира, вы должны передать конструктору три аргумента. (Четвертое значение — это член `total_val`; он вычисляется на основе `shares` и `share_val`, поэтому передавать его конструктору не понадобится.) Возможно, вы решите передать только значение члена `company` и установить остальные члены в нули; это можно сделать с использованием аргументов по умолчанию (см. главу 8). Таким образом, прототип будет выглядеть следующим образом:

```
// Прототип конструктора с несколькими аргументами по умолчанию
Stock(const string & co, long n = 0, double pr = 0.0);
```

Первый аргумент представляет собой указатель на строку, используемую для инициализации члена класса `company` типа `string`. Аргументы `n` и `pr` предоставляют значения для членов `shares` и `share_val`. Обратите внимание, что тип возвращаемого значения не указан. Прототип размещен в открытом разделе объявления класса.

Ниже приведен один из вариантов определения конструктора:

```
// Определение конструктора
Stock::Stock(const string & co, long n, double pr)
{
   company = co;
```

```
    if (n < 0)
    {
        std::cerr << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}
```

Это тот же код, который использовался ранее в настоящей главе для функции `acquire()`. Разница в том, что в данном случае программа автоматически вызовет конструктор при объявлении объекта.

Имена членов и имена параметров

Новички часто пытаются использовать имена переменных-членов класса в качестве имен аргументов в конструкторе, как показано в следующем примере:

```
// Так поступать нельзя!
Stock::Stock(const string & company, long shares, double share_val)
{
    ...
```

Это неверно. Аргументы конструктора не являются переменными-членами; они представляют значения, которые присваиваются членам класса. Таким образом, они должны иметь отличающиеся имена, иначе вы столкнетесь с непонятным кодом вроде такого:

```
shares = shares;
```

Одним из часто используемых способов, призванных помочь избежать этого, является использование префикса `m_` для идентификации данных-членов:

```
class Stock
{
private:
    string m_company;
    long m_chares;
    ...
```

Другой также часто применяемый способ заключается в применении суффикса в виде подчеркивания для имен членов:

```
class Stock
{
private:
    string company_;
    long shares_;
    ...
```

Воспользовавшись одним из соглашений, в качестве имен параметров в открытом интерфейсе можно использовать `company` и `shares`.

Использование конструкторов

Язык C++ предлагает два способа инициализации объектов с помощью конструктора. Первый — вызвать конструктор явно:

```
Stock food = Stock("World Cabbage", 250, 1.25);
```

Это устанавливает значение члена company объекта food равным строке "World Cabbage", значение shares равным 250 и т.д.

Второй способ — вызвать конструктор неявно:

```
Stock garment("Furry Mason", 50, 2.5);
```

Эта более компактная форма эквивалентна следующему явному вызову:

```
Stock garment = Stock("Furry Mason", 50, 2.5);
```

C++ использует конструктор класса всякий раз, когда вы создаете объект класса, даже если применяется операция new для динамического выделения памяти. Ниже показано, как использовать конструктор вместе с new:

```
Stock *pstock = new Stock("Electroshock Games",18,19.0);
```

Оператор, создающий объект Stock, инициализирует его значениями, переданными в аргументах, и присваивает адрес нового объекта указателю pstock. В этом случае объект не имеет имени, но для управления объектом можно применять указатель. Указатели на объекты будут обсуждаться в главе 11.

Конструкторы используются способом, отличным от всех остальных методов класса. Обычно объект применяется для вызова метода:

```
stock1.show();          // объект stock вызывает метод show()
```

Однако нельзя использовать объект для вызова конструктора, поскольку до тех пор, пока конструктор не завершит создание объекта, его не существует. Вместо того чтобы вызываться объектом, конструктор служит для создания объекта.

Конструкторы по умолчанию

Конструктор по умолчанию — это конструктор, который используется для создания объекта, когда не предоставлены явные инициализирующие значения. То есть это конструктор, который применяется для объявлений, подобных показанному ниже:

```
Stock fluffy_the_cat;    // используется конструктор по умолчанию
```

Но ведь в листинге 10.3 уже делалось это! Причина, по которой этот оператор работает, состоит в том, что если вы забудете о написании конструкторов, то C++ автоматически создаст конструктор по умолчанию. Это — неявная версия конструктора по умолчанию, который ничего не делает. Для класса Stock конструктор по умолчанию будет таким:

```
Stock::Stock() { }
```

В результате создается объект fluffy_the_cat с инициализированными членами, как в следующем операторе создается x без указания его значения:

```
int x;
```

То, что конструктор по умолчанию не имеет аргументов, отражает факт отсутствия значений в объявлении.

Любопытным моментом, имеющим отношение к конструктору по умолчанию, является то, что компилятор создает его, только если вы не определите ни одного собственного конструктора. После того, как вы определите хотя бы один конструктор класса, компилятор перестанет создавать конструктор по умолчанию. Если вы предоставите конструктор не по умолчанию вроде Stock(const string & co, long n, double pr), но не предложите собственную версию конструктора по умолчанию, то следующее объявление вызовет ошибку:

```
Stock stock1;           // невозможно с существующим конструктором
```

Причина такого поведения в том, что может понадобиться запрет создания неинициализированных объектов. Если же, однако, вы предпочитаете создавать объекты без явной инициализации, то должны будете определить собственный конструктор. Этот конструктор не имеет аргументов. Конструктор по умолчанию можно создать двумя способами. Один из них предусматривает указание значений по умолчанию для всех аргументов в существующем конструкторе:

```
Stock(const string & co = "Error", long n = 0, double pr = 0.0)
```

Второй способ — использование возможности перегрузки функций для определения второго конструктора без аргументов:

```
Stock();
```

Допускается наличие только одного конструктора по умолчанию, поэтому удостоверьтесь, что не создали их два. На самом деле обычно вы должны инициализировать объекты для гарантии того, что при создании объекта все члены получают известные и подходящие значения. Таким образом, пользовательский конструктор по умолчанию, как правило, обеспечивает явную инициализацию всех переменных-членов. Например, ниже показано, как можно определить конструктор по умолчанию для класса `Stock`:

```
Stock:: Stock ()              // конструктор по умолчанию
{
   company = "no name";
   shares = 0;
   share_val = 0.0;
   total_val = 0.0;
}
```

Совет

При проектировании класса обычно должен быть предусмотрен конструктор по умолчанию, который неявно инициализирует все переменные-члены класса.

После создания конструктора по умолчанию любым из двух способов (без аргументов или с предоставлением значений по умолчанию для всех аргументов) можно объявлять объектные переменные без явной инициализации:

```
Stock first;                  // вызывает конструктор по умолчанию неявно
Stock first = Stock();        // вызывает конструктор по умолчанию явно
Stock *prelief = new Stock;   // вызывает конструктор по умолчанию неявно
```

Однако вас не должна сбивать с толку неявная форма конструктора не по умолчанию:

```
Stock first("Concrete Conglomerate");// вызывает конструктор
Stock second();               // объявляет функцию
Stock third;                  // вызывает конструктор по умолчанию
```

Первое объявление из приведенных выше вызывает конструктор не по умолчанию — т.е. такой, который принимает аргументы. Второе объявление устанавливает, что `second()` — это функция, возвращающая объект `Stock`. При неявном вызове конструктора по умолчанию круглые скобки указываться не должны.

Деструкторы

В случае использования конструктора для создания объекта программа отслеживает этот объект до момента его исчезновения. В этот момент программа автоматически

вызывает специальную функцию-член с несколько пугающим названием *деструктор*. Деструктор призван очищать всяческий "мусор", поэтому он служит весьма полезной цели. Например, если в конструкторе используется операция `new` для выделения памяти, то деструктор должен обратиться к `delete` для ее освобождения. Конструктор нашего класса `Stock` не делает никаких причудливых действий наподобие вызова `new`, поэтому деструктору класса `Stock` делать нечего. В таком случае вы можете просто позволить компилятору сгенерировать неявный деструктор, который ничего не делает, что и имеет место в первой версии класса `Stock`. С другой стороны, полезно посмотреть, как объявляются и определяются деструкторы, поэтому давайте предусмотрим это в классе `Stock`.

Как и конструктор, деструктор имеет специальное имя. Оно формируется из имени класса и предваряющего его символа тильды (~). То есть деструктор для класса `Stock` называется `~Stock()`. Подобно конструктору, деструктор не имеет ни возвращаемого значения, ни объявляемого типа. Однако в отличие от конструктора, деструктор не должен иметь аргументы. Таким образом, прототип деструктора класса `Stock` выглядит следующим образом:

```
~Stock();
```

Поскольку деструктор `Stock` не имеет никаких обязанностей, его можно кодировать как функцию, которая ничего не делает:

```
Stock::~Stock()
{
}
```

Но просто для того, чтобы увидеть, когда вызывается конструктор, определим его следующим образом:

```
Stock::~Stock()
{
    cout << "Bye, " << company << "!\n";
}
```

Когда должен вызываться деструктор? Этим управляет компилятор. Обычно деструктор не должен явно вызываться в коде. (Исключение из этого правила описано в главе 12.) Если вы создаете статический объект класса, то его деструктор вызывается автоматически при завершении работы программы. Если вы создаете автоматический (локальный) объект класса, как в приведенном примере, то его деструктор вызывается автоматически, когда выполнение программы покидает блок кода, в котором определен объект. Если объект создается с использованием операции `new`, он размещается в свободной памяти, и его деструктор вызывается автоматически, когда вызывается `delete` для ее освобождения. И, наконец, программа может создавать временные объекты для обслуживания некоторых операций; в этом случае деструктор вызывается тогда, когда программа завершает пользование объектом.

Поскольку деструктор вызывается автоматически при уничтожении объекта класса, деструктор должен существовать. Если вы его не предусмотрите, компилятор неявно создаст конструктор по умолчанию и, если обнаружит код, который ведет к уничтожению объекта, также неявно создаст деструктор.

Усовершенствование класса `Stock`

Теперь необходимо включить конструкторы и деструктор в определение класса и методов. Учитывая важность добавления конструкторов, изменим имя `stock00.h` на `stock10.h`. Методы класса будут находиться в файле по имени `stock10.cpp`.

И, наконец, программа, использующая эти ресурсы, будет помещена в третий файл — usestok2.cpp.

Заголовочный файл

В листинге 10.4 показан заголовочный файл программы stock. К исходному объявлению класса здесь добавлены прототипы функций конструктора и деструктора. Также он отличается отсутствием функции acquire(), которая более не нужна, поскольку класс имеет конструкторы. В файле используется также прием #ifndef, описанный в главе 9, для защиты от многократного включения заголовочного файла.

Листинг 10.4. stock10.h

```
// stock10.h -- объявление класса Stock с добавленными конструкторами и деструктором
#ifndef STOCK10_H_
#define STOCK01_H_
#include <string>

class Stock
{

private:
    std::string company;
    long shares;
    double share_val;
    double total_val;
    void set_tot() { total_val = shares * share_val; }

public:
    // Два конструктора
    Stock();          // конструктор по умолчанию
    Stock(const std::string & co, long n = 0, double pr = 0.0);
    ~Stock();         // деструктор
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show();
};

#endif
```

Файл реализации

В листинге 10.5 приведены определения методов для разрабатываемой программы. Для предоставления программе необходимого объявления класса в нем включается файл stock10.h. (Вспомните, что помещение имени файла в двойные кавычки вместо угловых скобок заставляет компилятор искать его там же, где расположены файлы исходного кода.)

Кроме того, в листинге 10.5 включен заголовочный файл iostream для обеспечения поддержки ввода-вывода. В коде также демонстрируется использование объявлений using и уточненных имен (наподобие std::string) для обеспечения доступа к различным определениям из заголовочных файлов. В этом файле к предшествующим методам добавлены определения методов конструктора и деструктора. Чтобы помочь увидеть момент вызова метода, каждый из них отображает сообщение. Это не является обычным свойством конструкторов и деструкторов, однако позволяет сделать наглядным их использование классом.

Листинг 10.5. stock10.cpp

```
// stock10.cpp -- реализация класса Stock с добавленными конструкторами и деструктором
#include <iostream>
#include "stock10.h"

// Конструкторы (версии с выводом сообщений)
Stock::Stock()        // конструктор по умолчанию
{
    std::cout << "Default constructor called\n";
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}

Stock::Stock(const std::string & co, long n, double pr)
{
    std::cout << "Constructor using " << co << " called\n";
    company = co;
    if (n < 0)
    {
        std::cout << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}

// Деструктор класса
Stock::~Stock()      // деструктор класса, отображающий сообщение
{
    std::cout << "Bye, " << company << "!\n";
}

// Другие методы
void Stock::buy(long num, double price)
{
    if (num < 0)
    {
        std::cout << "Number of shares purchased can't be negative. "
                  << "Transaction is aborted.\n";
    }
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}
void Stock::sell(long num, double price)
{
    using std::cout;
    if (num < 0)
    {
        cout << "Number of shares sold can't be negative. "
             << "Transaction is aborted.\n";
    }
```

```
    else if (num > shares)
    {
        cout << "You can't sell more than you have! "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show()
{
    using std::cout;
    using std::ios_base;
    // Установка формата в #.###
    ios_base::fmtflags orig =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    std::streamsize prec = cout.precision(3);
    cout << "Company: " << company
         << " Shares: " << shares << '\n';
    cout << " Share Price: $" << share_val;
    // Установка формата в #.##
    cout.precision(2);
    cout << " Total Worth: $" << total_val << '\n';
    // Восстановление исходного формата
    cout.setf(orig, ios_base::floatfield);
    cout.precision(prec);
}
```

Файл клиентской программы

В листинге 10.6 представлена короткая программа для тестирования новых методов. Из-за того, что здесь просто используется класс `Stock`, код, приведенный в листинге, является клиентом класса `Stock`. Как и `stock10.cpp`, он включает файл `stock10.h` для доступа к объявлению класса. Эта программа демонстрирует работу конструкторов и деструктора. В ней также используются те же команды форматирования, что и в листинге 10.3. Для компиляции полной программы применяйте приемы, предназначенные для многофайловых программ, которые были описаны в главах 1 и 9.

Листинг 10.6. `usestock1.cpp`

```
//usestock1.cpp -- использование класса Stock
// Компилировать вместе с stock10.cpp
#include <iostream>
#include "stock10.h"

int main()
{
    {
        using std::cout;
```

```
    // Использование конструкторов для создания новых объектов
    cout << "Using constructors to create new objects\n";
    Stock stock1("NanoSmart", 12, 20.0);            // первый синтаксис
    stock1.show();
    Stock stock2 = Stock ("Boffo Objects", 2, 2.0);  // второй синтаксис
    stock2.show();

    // Присваивание stock1 объекту stock2
    cout << "Assigning stock1 to stock2:\n";
    stock2 = stock1;

    // Вывод stock1 и stock2
    cout << "Listing stock1 and stock2:\n";
    stock1.show();
    stock2.show();

    // Использование конструктора для сброса объекта
    cout << "Using a constructor to reset an object\n";
    stock1 = Stock("Nifty Foods", 10, 50.0);        // временный объект
    cout << "Revised stock1:\n";
    stock1.show();
    cout << "Done\n";
  }
  return 0;
}
```

В результате компиляции кода, представленного в листингах 10.4, 10.5 и 10.6, генерируется исполняемая программа. Ниже показано, как выглядит вывод этой программы, скомпилированной одним из компиляторов:

```
Using constructors to create new objects
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
  Share Price: $20.00  Total Worth: $240.00
Constructor using Boffo Objects called
Company: Boffo Objects  Shares: 2
  Share Price: $2.00  Total Worth: $4.00
Assigning stock1 to stock2:
Listing stock1 and stock2:
Company: NanoSmart  Shares: 12
  Share Price: $20.00  Total Worth: $240.00
Company: NanoSmart  Shares: 12
  Share Price: $20.00  Total Worth: $240.00
Using a constructor to reset an object
Constructor using Nifty Foods called
Bye, Nifty Foods!
Revised stock1:
Company: Nifty Foods  Shares: 10
  Share Price: $50.00  Total Worth: $500.00
Done
Bye, NanoSmart!
Bye, Nifty Foods!
```

Определенные компиляторы могут сгенерировать программу, вывод из которой содержит дополнительную строку:

```
Using constructors to create new objects
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
  Share Price: $20.00  Total Worth: $240.00
```

```
Constructor using Boffo Objects called
Bye, Boffo Objects!                    ← дополнительная строка
Company: Boffo Objects  Shares: 2
  Share Price: $2.00  Total Worth: $4.00
...
```

В приведенном ниже разделе "Замечания по программе" объясняется появление дополнительной строки "Bye, Boffo Objects!" в выводе из программы.

> **На заметку!**
>
> Вы наверняка заметили, что в листинге 10.6 присутствуют дополнительные фигурные скобки в начале и почти в конце функции `main()`. Срок существования автоматических переменных, таких как `stock1` и `stock2`, истекает, когда выполнение программы покидает блок, содержащий их определения. Без этих дополнительных фигурных скобок таким блоком являлось бы тело функции `main()`, поэтому деструкторы не были бы вызваны вплоть до полного завершения `main()`. В оконной среде это означает, что окно программы закроется перед вызовом двух деструкторов, и увидеть два последних сообщения не удастся. Но благодаря скобкам, два последних вызова деструкторов произойдут перед достижением оператора `return`, так что сообщения смогут отобразиться.

Замечания по программе

В листинге 10.6 оператор

```
Stock stock1("NanoSmart", 12, 20.0);
```

создает объект `Stock` по имени `stock1` и инициализирует его данные-члены указанными значениями:

```
Constructor using NanoSmart called
Company: NanoSmart  Shares: 12
```

Следующий оператор использует другой синтаксис для создания и инициализации объекта `stock2`:

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
```

Стандарт C++ позволяет компилятору выполнять второй синтаксис двумя способами. Один из них характеризуется тем же поведением, что и в случае первого синтаксиса:

```
Constructor using Boffo Objects called
Company: Boffo Objects  Shares: 2
```

Второй способ реализации заключается в вызове конструктора для создания временного объекта, который затем копируется в `stock2`. После этого временный объект уничтожается. Если компилятор использует этот вариант, то для временного объекта вызывается деструктор, что приводит к следующему выводу:

```
Constructor using Boffo Objects called
Bye, Boffo Objects!
Company: Boffo Objects  Shares: 2
```

Компилятор, который обеспечивает генерацию такого вывода, обычно освобождает временный объект немедленно, но может быть и так, что он ожидает некоторое время — в этом случае сообщение из деструктора появится позже.

Приведенный ниже оператор иллюстрирует возможность присваивания одного объекта другому объекту того же типа:

```
stock2 = stock1;  // присваивание объекта
```

Как и в случае присваивания структур, присваивание объектов класса по умолчанию копирует члены одного объекта в другой. В этом случае исходное содержимое stock2 перезаписывается.

На заметку!

В случае присваивания одного объекта другому объекту того же класса по умолчанию C++ копирует содержимое каждого члена данных исходного объекта в соответствующий член данных другого объекта.

Конструктор можно использовать не только для инициализации нового объекта. Например, в функции main() присутствует такой оператор:

```
stock1 = Stock("Nifty Foods", 10, 50.0);
```

Объект stock1 уже существует. Следовательно, вместо инициализации объекта stock1 показанный оператор присваивает ему новые значения. Это делается за счет создания конструктором нового временного объекта и последующего копирования его содержимого в stock1. Затем программа уничтожает временный объект, вызывая его деструктор, что и иллюстрирует следующий аннотированный вывод:

```
Using a constructor to reset an object
Constructor using Nifty Foods called        ← временный объект создан
Bye, Nifty Foods!                           ← временный объект уничтожен
Revised stock1:
Company: Nifty Foods  Shares: 10            ← данные скопированы в stock1
  Share Price: $50.00  Total Worth: $500.00
```

Некоторые компиляторы могут освобождать временный объект позже, откладывая вызов деструктора.

В конце работы программа отображает следующие сообщения:

```
Done
Bye, NanoSmart!
Bye, Nifty Foods!
```

Когда функция main() завершает работу, ее локальные переменные (stock1 и stock2) перестают существовать. Поскольку такие автоматические переменные размещаются в стеке, последний созданный объект удаляется первым, а первый созданный — последним. (Вспомним, что строка "NanoSmart" находилась изначально в stock1, но позже была перенесена в stock2, а объект stock1 был сброшен в "Nifty Foods".)

Вывод программы демонстрирует фундаментальную разницу между следующими двумя операторами:

```
Stock stock2 = Stock ("Boffo Objects", 2, 2.0);
stock1 = Stock("Nifty Foods", 10, 50.0);  // временный объект
```

Первый из этих операторов вызывает инициализацию; он создает объект с указанным значением, и может создавать либо не создавать временный объект. Второй оператор вызывает присваивание. Использование конструктора в операции присваивания в таком виде всегда служит причиной создания временного объекта перед выполнением собственно присваивания.

Совет

Если устанавливать значения объекта можно как с помощью инициализации, так и присваивания, выбирайте вариант инициализации. Обычно это более эффективно.

Списковая инициализация C++11

Можно ли в C++11 использовать синтаксис списковой инициализации для классов? Да, можно; для этого потребуется предоставить в фигурных скобках содержимое, соответствующее списку аргументов конструктора:

```
Stock hot_tip = {"Derivatives Plus Plus", 100, 45.0};
Stock jock {"Sport Age Storage, Inc"};
Stock temp {};
```

Списки в фигурных скобках в первых двух объявлениях соответствуют следующему конструктору:

```
Stock::Stock(const std::string & co, long n = 0, double pr = 0.0);
```

Таким образом, этот конструктор будет использоваться для создания двух объектов. В случае объекта `jock` для второго и третьего аргументов будут применяться значения по умолчанию — `0` и `0.0`. Третье объявление соответствует конструктору по умолчанию, поэтому объект `temp` будет создан с его помощью.

Вдобавок C++11 предлагает класс по имени `std::initializer_list`, который может использоваться в качестве типа для параметра функции или метода. Этот класс представляет список произвольной длины, все элементы которого имеют один и тот же тип или могут быть преобразованы к одному и тому же типу. Мы вернемся к этой теме в главе 16.

Функции-члены `const`

Рассмотрим следующий фрагмент кода:

```
const Stock land = Stock("Kludgehorn Properties");
land.show();
```

Компилятор современного языка C++ не должен принять вторую строку. Почему? Причина в том, что код `show()` не гарантирует того, что он не изменит объект, который из-за объявления как `const` меняться не должен. Вы должны предварительно позаботиться о решении этой проблемы, объявив аргумент функции как ссылку `const` или указатель на `const`. Однако здесь существует синтаксическая сложность: в методе `show()` нет аргументов, которые можно было бы квалифицировать как `const`. Вместо них используемый объект неявно задан вызовом этого метода. Необходим новый синтаксис, который укажет на то, что функция-член не будет модифицировать объект. Решение, предлагаемое C++, заключается в помещении ключевого слова `const` после скобок функции. То есть объявление метода `show()` должно выглядеть следующим образом:

```
void show() const;          // обещает не изменять вызываемый объект
```

Аналогично начало определения функции должно выглядеть так, как показано ниже:

```
void Stock::show() const    // обещает не изменять вызываемый объект
```

Функции класса, объявленные и определенные подобным образом, называются константными функциями-членами. Точно так же, как константные ссылки и указатели, где это необходимо, используются в качестве формальных аргументов функций, вы должны делать методы класса константными всегда, когда они не модифицируют объект, с которым работают. Отныне мы будем следовать этому правилу.

Обзор конструкторов и деструкторов

Теперь, после ознакомления с рядом примеров конструкторов и деструкторов, сделаем паузу и подведем некоторые итоги. Ниже приводится краткий обзор этих методов.

Конструктор — это специальная функция-член класса, которая вызывается всякий раз при создании объекта данного класса. Конструктор класса имеет то же имя, что и класс, но благодаря возможностям перегрузки функций, существует возможность создавать более одного конструктора с одним и тем же именем и разным набором аргументов. Кроме того, конструктор не имеет объявленного типа. Обычно конструктор используется для инициализации членов объекта класса. Ваша инициализация должна соответствовать списку аргументов конструктора. Например, предположим, что класс `Bozo` имеет следующий прототип для конструктора:

```
Bozo(const char * fname, const char * lname);      // прототип конструктора
```

В этом случае его можно использовать для инициализации объекта следующим образом:

```
Bozo bozetta = Bozo("Bozetta", "Biggens");         // основная форма
Bozo fufu("Fufu", "O'Dweeb");                      // сокращенная форма
Bozo *pc = new Bozo("Popo", "Le Peu");             // динамический объект
```

В C++11 можно взамен применять списковую инициализацию:

```
Bozo bozetta = {"Bozetta", "Biggens"};             // C++11
Bozo fufu{"Fufu", "O'Dweeb"};                      // C++11
Bozo *pc = new Bozo{"Popo", "Le Peu"};             // C++11
```

Когда конструктор имеет только один аргумент, он вызывается в случае инициализации объекта значением, которое имеет тот же тип, что и аргумент конструктора. Например, предположим, что существует следующий прототип конструктора:

```
Bozo(int age);
```

Тогда в коде можно использовать любую из следующих форм инициализации объекта:

```
Bozo dribble = Bozo(44);  // первичная форма
Bozo roon(66);            // вторичная форма
Bozo tubby = 32;      // специальная форма для конструктора с одним аргументом
```

Фактически третий пример является новым, и это удобный момент, чтобы сказать о нем. В главе 11 упомянут способ отключения этого средства, поскольку оно может привести к неприятным сюрпризам.

Внимание!

Конструктор, который принимает один аргумент, позволяет использовать синтаксис присваивания для инициализации объекта значением:

имяКласса объект = значение;

Эта возможность может привести к возникновению проблем, но ее можно заблокировать, как будет показано в главе 11.

Конструктор по умолчанию не имеет аргументов и используется, когда вы создаете объект без явной его инициализации. Если вы не предоставляете ни одного конструктора, то компилятор создаст конструктор по умолчанию самостоятельно. В противном случае вы обязаны определить собственный конструктор по умолчанию. Он

может либо не иметь аргументов, либо предусматривать значения по умолчанию для всех аргументов:

```
Bozo();                                    // прототип конструктора по умолчанию
Bistro(const char *s = "Chez Zero");       // значение по умолчанию для класса
Bistro
```

Программа использует конструкторы по умолчанию для неинициализированных объектов:

```
Bozo bibi;                  // используется конструктор по умолчанию
Bozo *pb = new Bozo;        // используется конструктор по умолчанию
```

Подобно тому, как при создании объекта вызывается конструктор, деструктор вызывается при его уничтожении. Для класса допускается наличие только одного деструктора. Он не имеет возвращаемого типа (даже `void`), не имеет аргументов, и его имя состоит из имени класса с предшествующей тильдой (~). Например, деструктор класса `Bozo` имеет следующий прототип:

```
~Bozo();                    // деструктор класса
```

Деструкторы классов, в которых используется операция `delete`, становятся необходимыми, когда в конструкторах классов применяется операция `new`.

Изучение объектов: указатель this

С классом `Stock` можно делать кое-что еще. До сих пор каждая функция-член класса имела дело только с одним объектом: тем, который ее вызывал. Однако иногда методу может понадобиться иметь дело с двумя объектами и для этого обращаться к любопытному указателю по имени `this`. Давайте посмотрим, когда может понадобиться `this`.

Несмотря на то что объявление класса `Stock` включает в себя отображение данных, все же ему недостает аналитических возможностей. Например, если взглянете на вывод функции `show()`, то вы сможете сказать, какая из ваших долей обладает наибольшим пакетом акций, но программа не сможет дать ответ на этот вопрос, поскольку не имеет прямого доступа к `total_val`. Самый простой способ сообщить программе о хранимых данных — это предусмотреть методы, возвращающие эти данные. Обычно для этого применяется встроенный код, как в следующем примере:

```
class Stock
{
private:
    ...
    double total_val;
    ...
public:
    double total() const { return total_val; }
    ...
};
```

Это определение делает `total_val` доступным в программе только для чтения. То есть метод `total()` можно использовать для получения этого значения, но класс не предоставляет метода для его переустановки. (Другие методы, такие как `buy()`, `sell()` и `update()`, модифицируют `total_val` в качестве побочного эффекта от переустановки значений членов `shares` и `share_val`.)

За счет добавления этой функции к объявлению класса можно позволить программе исследовать последовательности пакетов акций для поиска наиболее крупного из них. Однако можно воспользоваться другим подходом, который поможет разобраться с указателем this. Подход заключается в определении функции-члена, которая будет просматривать два объекта Stock и возвращать ссылку на больший из них. Попытка реализовать эту идею вызывает некоторые интересные вопросы, которые мы сейчас рассмотрим.

Во-первых, как написать функцию, работающую с двумя объектами с целью их сравнения? Предположим, например, что ее решено назвать topval(). После этого вызов stock1.topval() обращается к данным объекта stock1, в то время как stock2.topval() — к данным объекта stock2. Если нужно, чтобы метод сравнивал два объекта, второй объект потребуется передать в виде аргумента. Для эффективности его можно передавать по ссылке. Это значит, метод topval() должен принимать аргумент типа const Stock &.

Во-вторых, как результат метода будет передаваться в вызывающую программу? Самый прямой путь — заставить метод возвращать ссылку на объект, который имеет большее значение total_val. Таким образом, метод сравнения двух объектов будет иметь следующий прототип:

```
const Stock & topval(const Stock & s) const;
```

Эта функция имеет неявный доступ к одному объекту и явный — ко второму, и она возвращает ссылку на один из двух объектов. Слово const внутри скобок указывает, что функция не будет модифицировать объект, к которому получает явный доступ, а слово const, которое следует за скобками, устанавливает, что функция не будет изменять объект, на который ссылается неявно. Поскольку функция возвращает ссылку на один из const-объектов, тип ее возврата также является ссылкой const.

Предположим, что вы хотите сравнить два объекта Stock — stock1 и stock2 — и присвоить объекту top тот из них, который имеет большее значение total_val. Для этого можно воспользоваться любым из следующих двух операторов:

```
top = stock1.topval(stock2);
top = stock2.topval(stock1);
```

Первая форма обращается к stock1 неявно, а к stock2 — явно, в то время как вторая — наоборот (рис. 10.3). В любом случае метод сравнивает два объекта и возвращает ссылку на тот, который имеет большее значение total_val.

В действительности такая нотация несколько запутывает. Было бы понятнее, если бы удалось каким-то образом использовать операцию > для сравнения двух объектов. Это можно сделать с помощью перегрузки операций, которая обсуждается в главе 11.

Между тем, пока рассмотрим реализацию topval(). Она порождает небольшую проблему. Вот часть реализации, иллюстрирующая проблему:

```
const Stock & Stock::topval(const Stock & s) const
{
  if (s.total_val > total_val)
       return s;          // объект-аргумент
  else
       return ?????;      // вызывающий объект
}
```

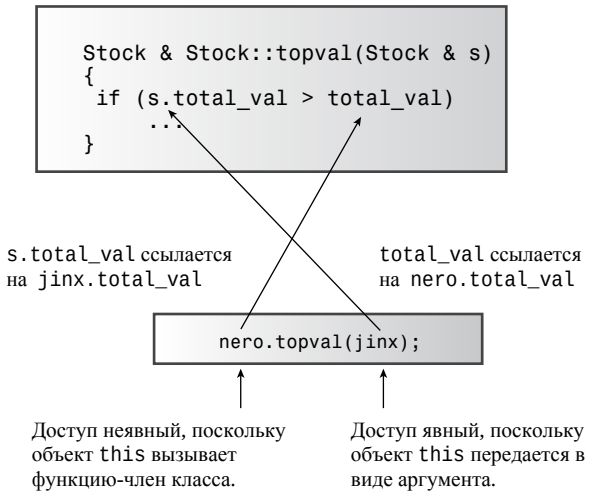


***Рис. 10.3.** Доступ к двум объектам из функции-члена*

Здесь `s.total_val` — это суммарное значение объекта, переданное в виде аргумента, а `total_val` — суммарное значение объекта, которому сообщение передается. Если `s.total_val` больше `total_val`, то функция возвращает `s`. В противном случае она возвратит объект, использованный для вызова метода. (В терминологии ООП — это объект, которому передано сообщение `topval`.) Существует одна проблема: на чем вызывать объект? Если сделать вызов `stock1.topval(stock2)`, то `s` — это ссылка на `stock2` (т.е. псевдоним для `stock2`), но псевдонима для `stock1` не существует.

Решение этой проблемы, которое предлагает C++, заключается в применении специального указателя `this`. Он указывает на объект, который использован для вызова функции-члена. (Обычно `this` передается методу в виде скрытого аргумента.) Таким образом, вызов `stock1.topval(stock2)` устанавливает значение `this` равным адресу объекта `stock1` и делает его доступным методу `topval()`. Аналогичным образом, вызов функции `stock2.topval(stock1)` устанавливает значение `this` равным адресу объекта `stock2`. Вообще все методы класса получают указатель `this`, равный адресу объекта, который вызвал метод. Фактически `total_val` внутри `total()` является сокращенной нотацией `this->total_val`. (Вспомните из главы 4, что операция `->` используется для доступа к членам структуры через указатель на нее. То же самое верно и для членов класса.) Обратите внимание на рис. 10.4.

На заметку!

Каждая функция-член, включая конструкторы и деструкторы, имеет указатель `this`. Специфическим свойством `this` является то, что он указывает на вызывающий объект. Если метод нуждается в получении ссылки на вызвавший объект в целом, он может использовать выражение `*this`. Применение квалификатора `const` после скобок с аргументами заставляет трактовать `this` как указатель на `const`; в этом случае вы не можете использовать `this` для изменения значений объекта.

Однако то, что необходимо вернуть из метода — это не `this`, поскольку `this` представляет собой адрес объекта. Вам нужно вернуть сам объект, а это обозначается выражением `*this`. (Вспомните, что применение операции разыменования `*` к указателю дает значение, на которое он указывает.)

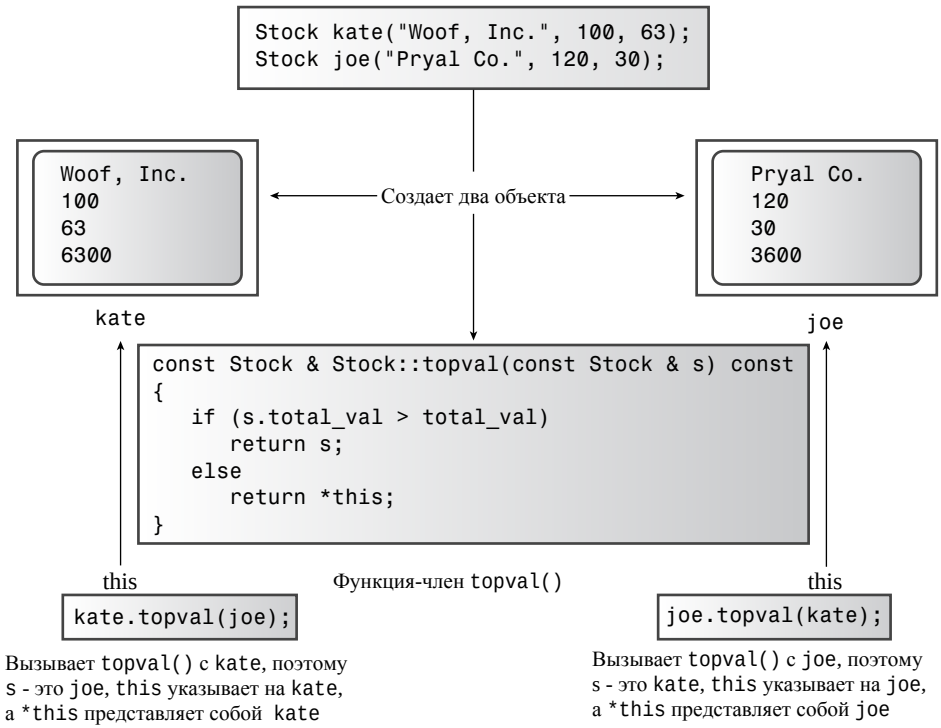


Рис. 10.4. *this указывает на вызвавший объект*

Теперь можно завершить определение метода, используя `*this` в качестве псевдонима вызвавшего объекта:

```
const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
      return s;          // объект-аргумент
   else
      return *this;      // вызывающий объект
}
```

Тот факт, что возвращаемое значение представляет собой ссылку, означает, что возвращаемый объект является тем же самым объектом, который вызвал данный метод, а не копией, переданной механизмом возврата. В листинге 10.7 приведен новый заголовочный файл.

Листинг 10.7. `stock20.h`

```
// stock20.h — дополненная версия
#ifndef STOCK20_H_
#define STOCK20_H_
#include <string>

class Stock
{
private:
   std::string company;
   int shares;
   double share_val;
```

```
    double total_val;
    void set_tot() { total_val = shares * share_val; }
public:
    Stock();          // конструктор по умолчанию
    Stock(const std::string & co, long n = 0, double pr = 0.0);
    ~Stock();         // деструктор
    void buy(long num, double price);
    void sell(long num, double price);
    void update(double price);
    void show()const;
    const Stock & topval(const Stock & s) const;
};
#endif
```

В листинге 10.8 показан измененный файл с методами класса. Он включает в себя новый метод `topval()`. Также теперь, когда вы ознакомились с работой конструкторов и деструкторов, в листинге 10.8 они заменены версиями, которые не выводят никаких сообщений.

Листинг 10.8. `stocks20.cpp`

```
// stock20.cpp -- дополненная версия
#include <iostream>
#include "stock20.h"
// Конструкторы
Stock::Stock()       // конструктор по умолчанию
{
    company = "no name";
    shares = 0;
    share_val = 0.0;
    total_val = 0.0;
}
Stock::Stock(const std::string & co, long n, double pr)
{
    company = co;
    if (n < 0)
    {
        std::cout << "Number of shares can't be negative; "
                  << company << " shares set to 0.\n";
        shares = 0;
    }
    else
        shares = n;
    share_val = pr;
    set_tot();
}
// Деструктор
Stock::~Stock()      // деструктор, не выводящий сообщений
{
}
// Другие методы
void Stock::buy(long num, double price)
{
    if (num < 0)
    {
        std::cout << "Number of shares purchased can't be negative. "
                  << "Transaction is aborted.\n";
    }
```

```
    else
    {
        shares += num;
        share_val = price;
        set_tot();
    }
}

void Stock::sell(long num, double price)
{
    using std::cout;
    if (num < 0)
    {
        cout << "Number of shares sold can't be negative. "
             << "Transaction is aborted.\n";
    }
    else if (num > shares)
    {
        cout << "You can't sell more than you have! "
             << "Transaction is aborted.\n";
    }
    else
    {
        shares -= num;
        share_val = price;
        set_tot();
    }
}

void Stock::update(double price)
{
    share_val = price;
    set_tot();
}

void Stock::show() const
{
    using std::cout;
    using std::ios_base;
    // Установка формата в #.###
    ios_base::fmtflags orig =
        cout.setf(ios_base::fixed, ios_base::floatfield);
    std::streamsize prec = cout.precision(3);
    cout << "Company: " << company
         << " Shares: " << shares << '\n';
    cout << " Share Price: $" << share_val;
    // Установка формата в #.##
    cout.precision(2);
    cout << " Total Worth: $" << total_val << '\n';
    // Восстановление исходного формата
    cout.setf(orig, ios_base::floatfield);
    cout.precision(prec);
}

const Stock & Stock::topval(const Stock & s) const
{
    if (s.total_val > total_val)
        return s;
    else
        return *this;
}
```

Разумеется, возникает желание проверить работу указателя this, и лучший способ сделать это — использовать новый метод в программе с массивом объектов, что и делается в следующем разделе.

Массив объектов

Часто, как и в примерах со Stock, требуется создавать несколько объектов одного класса. Можно создать отдельные объектные переменные, как это делалось до сих пор в примерах настоящей главы, но больше смысла будет в создании массива объектов. Это может выглядеть подобно прыжку в неизвестность, но фактически массив объектов объявляется таким же способом, как и массивы любых стандартных типов:

```
Stock mystuff[4];         // создание массива из 4 объектов Stock
```

Вспомните, что программа всегда вызывает конструктор по умолчанию, когда создает объекты класса без явной инициализации. Такое объявление требует либо отсутствия у класса явно определенных конструкторов (при этом используются неявные, ничего не делающие конструкторы), либо, как и в представленном случае — чтобы был явно определен конструктор по умолчанию. Каждый элемент — mystuff[0], mystuff[1] и т.д. — является объектом класса Stock, а потому может применяться с методами Stock:

```
mystuff[0].update();      // применяет update() к первому элементу
mystuff[3].show();        // применяет show() к 4-му элементу
const Stock * tops = mystuff[2].topval(mystuff[1]);
   // сравнивает 2-й и 3-й элементы и устанавливает tops
   // в указатель на тот из них, у которого больше значение total_val
```

Для инициализации элементов массива можно использовать конструктор. В этом случае необходимо вызывать конструктор для каждого индивидуального элемента:

```
const int STKS = 4;
Stock stocks[STKS] = {
   Stock("NanoSmart", 12.5, 20),
   Stock("Boffo Objects", 200, 2.0),
   Stock("Monolithic Obelisks", 130, 3.25),
   Stock("Fleep Enterprises", 60, 6.5)
};
```

В приведенном коде применяется стандартная форма инициализации массива: разделенный запятой список значений, заключенный в фигурные скобки. В таком случае каждое значение представлено вызовом метода конструктора. Если класс имеет более одного конструктора, для разных элементов можно использовать разные конструкторы:

```
const int STKS = 10;
Stock stocks[STKS] = {
   Stock("NanoSmart", 12.5, 20),
   Stock(),
   Stock("Monolithic Obelisks", 130, 3.25),
};
```

В коде элементы stocks[0] и stocks[2] инициализируются с помощью конструктора Stock(const string & co, long n, double pr), а stock[1] — посредством конструктора Stock(). Поскольку такое объявление инициализирует массив только частично, оставшиеся семь членов инициализируются конструктором по умолчанию.

В листинге 10.9 эти принципы применяются в короткой программе, которая инициализирует четыре элемента массива, отображает их содержимое и проверяет элементы в поисках того, который имеет наибольшее значение `total_val`. Поскольку `total()` сравнивает только два объекта за раз, для просмотра всего массива в программе используется цикл `for`. Для отслеживания элемента с наибольшим значением `total_val` применяется указатель на `Stock`. Код в этом листинге использует заголовочный файл из листинга 10.7 и файл методов из листинга 10.8.

Листинг 10.9. `usestock2.cpp`

```
// usestok2.cpp -- использование класса Stock
// Компилировать вместе с stock20.cpp
#include <iostream>
#include "stock20.h"
const int STKS = 4;
int main()
{
   // Создание массива инициализированных объектов
   Stock stocks[STKS] = {
     Stock("NanoSmart", 12, 20.0),
     Stock("Boffo Objects", 200, 2.0),
     Stock("Monolithic Obelisks", 130, 3.25),
     Stock("Fleep Enterprises", 60, 6.5)
   };
   std::cout << "Stock holdings:\n";
   int st;
   for (st = 0; st < STKS; st++)
      stocks[st].show();
   // Установка указателя н первый элемент
   const Stock * top = &stocks[0];
   for (st = 1; st < STKS; st++)
      top = &top->topval(stocks[st]);
   // Теперь top указывает на самый ценный пакет акций
   std::cout << "\nMost valuable holding:\n";
   top->show();
   return 0;
}
```

Ниже показан вывод программы из листинга 10.9:

```
Stock holdings:
Company: NanoSmart  Shares: 12
  Share Price: $20.000  Total Worth: $240.00
Company: Boffo Objects  Shares: 200
  Share Price: $2.000  Total Worth: $400.00
Company: Monolithic Obelisks  Shares: 130
  Share Price: $3.250  Total Worth: $422.50
Company: Fleep Enterprises  Shares: 60
  Share Price: $6.500  Total Worth: $390.00
Most valuable holding:
Company: Monolithic Obelisks  Shares: 130
  Share Price: $3.250  Total Worth: $422.50
```

Относительно листинга 10.9 следует отметить один момент: бо́льшая часть работы приходится на проектирование класса. Когда оно завершено, написание программы становится достаточно простым.

Между прочим, знание об указателе this позволяет заглянуть "за кулисы" C++. Например, исходная реализация для Unix использовала утилиту cfront, которая выполняла преобразование программ на C++ в программы на C. Для поддержки определений методов все, что нужно было сделать – это преобразовать определение метода C++ вроде такого:

```
void Stock::show() const
{
   cout << "Company: " << company
        << " Shares: " << shares << '\n'
        << " Share Price: $" << share_val
        << " Total Worth: $" << total_val << '\n';
}
```

в следующий код на языке C:

```
void show(const Stock * this)
{
   cout << "Company: " << this->company
        << " Shares: " << this->shares << '\n'
        << " Share Price: $" << this->share_val
        << " Total Worth: $" << this->total_val << '\n';
}
```

То есть квалификатор Stock:: преобразуется в аргумент функции, который представляет собой указатель на Stock, после чего этот указатель используется для доступа к членам класса.

Аналогичным образом преобразуются вызовы функций наподобие следующего:

```
top.show();
```

в такой вид:

```
show(&top);
```

В той же манере указателю this присваивается адрес вызывающего объекта. (Реальные детали этого процесса могут оказаться более сложными.)

Область видимости класса

В главе 9 обсуждались глобальная (на уровне файла) и локальная (на уровне блока) область видимости. Вспомните, что переменную с глобальной областью видимости можно использовать повсюду в файле, в котором она определена, в то время как переменная с локальной областью видимости является локальной по отношению к блоку, содержащему ее определение. Имена функций также могут иметь глобальную область видимости, но никогда – локальную. Классы C++ вводят новую разновидность области видимости – область видимости класса.

Область видимости класса применима к именам, определенным в классе, таким как имена данных-членов и функций-членов класса. Сущности, имеющие область видимости класса, известны внутри класса, но не известны за его пределами. Таким образом, одни и те же имена членов класса можно без конфликтов использовать в разных классах. Например, член shares класса Stock отличается от члена shares класса JobRide. Кроме того, область видимости класса означает, что вы не можете непосредственно обращаться к членам класса из внешнего мира. Это правило действует даже для открытых функций-членов. То есть для вызова открытой функции-члена должен использоваться объект:

```
Stock sleeper("Exclusive Ore", 100, 0.25);  // создание объекта
sleeper.show();           // использование объекта для вызова функции-члена
show();                   // неверно -- вызывать метод напрямую нельзя
```

Подобным же образом при определении функций-членов должна применяться операция разрешения контекста:

```
void Stock::update(double price)
{
   ...
}
```

Короче говоря, в пределах объявления класса или определения функции-члена можно использовать неуточненные (короткие) имена членов, как в ситуации, когда `sell()` вызывает функцию-член `set_tot()`. Имя конструктора распознается при вызове потому, что оно совпадает с именем класса. В противном случае должна применяться прямая операция членства (`.`), косвенная операция членства (`->`) или операция разрешения контекста (`::`), в зависимости от контекста, в котором используется имя члена класса. В следующем фрагменте кода иллюстрируется получение доступа к идентификаторам с областью видимости класса:

```
class Ik
{
private:
   int fuss;                  // fuss имеет область видимости класса
public:
   Ik(int f = 9) { fuss = f; } // fuss находится в области видимости
   void ViewIk() const;       // ViewIk имеет область видимости класса
};
void Ik::ViewIk() const       // Ik:: помещает ViewIk в область видимости Ik
{
  cout << fuss << endl; // fuss находится в области видимости внутри метода класса
}
...
int main()
{
   Ik * pik = new Ik;
   Ik ee = Ik(8);     // конструктор находится в области видимости,
                      // поскольку имеет имя класса
   ee.ViewIk();       // объект класса переносит ViewIk в область видимости
   pik->ViewIk();     // указатель на Ik переносит ViewIk в область видимости
   ...
```

Константы с областью видимости класса

Иногда хорошо бы иметь символические константы с областью видимости класса. Например, объявление класса может использовать литерал `12` для указания размера массива. Поскольку одна и та же константа применяется для всех объектов, было бы неплохо создать единственную константу, разделяемую всеми объектами. На первый взгляд, может показаться, что решается это следующим образом:

```
class Bakery
{
private:
   const int Months = 12;  // объявление константы? НЕ УДАСТСЯ
   double costs[Months];
   ...
```

Но это не работает, потому что объявление класса описывает, как выглядит объект, но не создает объекта. Следовательно, до тех пор, пока вы не создадите объект, хранить это значение негде. (На самом деле в C++11 предоставляется средство инициализации членов, но не таким способом, который бы позволил объявлять массив, как показано выше; мы вернемся к этой теме в главе 12.) Однако существует пара других способов достичь желаемой цели.

Первый способ заключается в том, чтобы объявить внутри класса перечисление. Такое перечисление имеет область видимости класса, поэтому его можно использовать в рамках класса как символическое имя для целочисленной константы. То есть начало объявления класса `Bakery` может выглядеть следующим образом:

```
class Bakery
{
private:
   enum {Months = 12};
   double costs[Months];
   ...
```

Обратите внимание, что такое объявление перечисления не создает переменную-член класса. То есть каждый индивидуальный объект не содержит его в себе. Вместо этого `Months` становится просто символическим именем, которое компилятор заменяет числом 12, когда встречает его в коде внутри области видимости класса.

Поскольку класс `Bakery` использует перечисление просто для создания символической константы, без намерения создавать переменные типа перечисления, то нет необходимости предоставлять дескриптор перечисления. Между прочим, во многих реализациях класс `ios_base` делает нечто подобное в своем разделе `public`; так объявлены идентификаторы вроде `ios_base::fixed`. Здесь `fixed` – обычно перечисление, определенное в классе `ios_base`.

В C++ имеется и второй способ определения константы в классе – с использованием ключевого слова `static`:

```
class Bakery
{
private:
   static const int Months = 12;
   double costs[Months];
   ...
```

Это создает одиночную константу по имени `Months`, хранящуюся вместе с остальными статическими переменными, а не в каждом объекте. Это значит, что существует только одна константа `Months`, которая разделяется между всеми объектами `Stock`. В главе 12 статические члены класса рассматриваются более подробно. В C++98 этот прием можно применять только для объявления статических констант с целыми и перечислимыми значениями. Однако подобным образом в C++98 невозможно хранить константу типа `double`. В C++11 это ограничение снято.

Перечисления с областью видимости (C++11)

С традиционными перечислениями связан ряд проблем. Одна из них состоит в том, что перечислители из двух разных определений `enum` могут конфликтовать друг с другом. Предположим, что в проекте требуется работать с яйцами (egg) и футболками (T-shirt). Можно попробовать следующие определения:

```
enum egg {Small, Medium, Large, Jumbo};
enum t_shirt {Small, Medium, Large, Xlarge};
```

Это работать не будет, потому что члены по имени `Small` в перечислениях `egg` и `t_shirt` будут находиться в одной и той же области видимости, вызывая конфликт имен. В C++11 предлагается новая форма перечисления, которая позволяет избежать этой проблемы за счет указания для перечислителей области видимости класса. Объявления в такой форме выглядят следующим образом:

```
enum class egg {Small, Medium, Large, Jumbo};
enum class t_shirt {Small, Medium, Large, Xlarge};
```

В качестве альтернативы вместо ключевого слова `class` можно использовать `struct`. В любом случае теперь понадобится указывать имя `enum` для уточнения перечислителя:

```
egg choice = egg::Large;        // перечислитель Large из перечисления egg
t_shirt Floyd = t_shirt::Large; // перечислитель Large из перечисления t_shirt
```

За счет наличия области видимости класса, перечислители из разных определений `enum` больше не имеют потенциальных конфликтов имен, так что работу над проектом можно продолжать. Для перечислений с областью видимости в C++11 также усилена безопасность типов. Обычные перечисления автоматически преобразуются в целочисленные типы в ряде ситуаций, таких как присваивание переменной `int` либо использование в выражении сравнения, но перечисления с областью видимости не поддерживают неявных преобразований в целочисленные типы:

```
enum egg_old {Small, Medium, Large, Jumbo};        // без области видимости
enum class t_shirt {Small, Medium, Large, Xlarge}; // с областью видимости
egg_old one = Medium;                              // без области видимости
t_shirt rolf = t_shirt::Large;                     // с областью видимости

int king = one;  // неявное преобразование для перечисления без области видимости
int ring = rolf; // не разрешено, неявное преобразование типа не поддерживается

if (king < Jumbo)                // разрешено
   std::cout << "Jumbo converted to int before comparison.\n";

if (king < t_shirt::Medium)    // не разрешено
   std::cout << "Not allowed: < not defined for scoped enum.\n";
```

Однако при необходимости можно выполнять явное преобразование типа:

```
int Frodo = int(t_shirt::Small); // Frodo устанавливается в 0
```

Перечисления представляются некоторым лежащим в основе целочисленным типом, и в C98 выбор такого типа возлагается на реализацию. Следовательно, структура, содержащая перечисление, в различных системах может иметь разные размеры. В C++11 эта зависимость для перечислений с областью видимости ликвидирована. По умолчанию лежащим в основе типом для перечислений с областью видимости C++11 является `int`. Более того, доступен синтаксис для указания другого лежащего в основе типа:

```
// Лежащим в основе типом для pizza является short
enum class : short pizza {Small, Medium, Large, XLarge};
```

Конструкция `: short` указывает, что лежащим в основе типом будет `short`. Лежащий в основе тип должен быть целочисленным. В C++11 с помощью этого же синтаксиса можно задать лежащий в основе тип для перечисления без области видимости, но если тип не указан, то он будет выбран компилятором в зависимости от реализации.

Абстрактные типы данных

Класс `Stock` довольно специфичен. Однако программисты часто определяют классы для представления более общих концепций. Например, использование классов — хороший способ реализации того, что специалисты в области вычислительной техники называют *абстрактным типом данных* (abstract data type — ADT). Как и можно было предположить, ADT описывает данные в общей манере, без деталей, связанных с языком или реализацией. Рассмотрим для примера стек. Используя стек, данные можно сохранять так, что они всегда будут добавляться или удаляться с его вершины. Например, программы на C++ применяют стек для управления автоматическими переменными. Когда новые переменные создаются, они добавляются на вершину стека, а когда уничтожаются, то удаляются из нее.

Давайте посмотрим на свойства стека в абстрактном смысле. Прежде всего, стек содержит множество элементов. (Это свойство делает его *контейнером* — т.е. еще более общей абстракцией.) Вдобавок стек характеризуется операциями, которые на нем можно выполнять:

- создание пустого стека;
- добавление элемента в вершину стека (т.е. *заталкивание* (push) элемента);
- удаление элемента из вершины стека (т.е. *выталкивание* (pop) элемента);
- проверка, полон ли стек;
- проверка, пуст ли стек.

Это описание может быть сопоставлено с объявлением класса, в котором общедоступные функции-члены предоставляют интерфейс, реализующий операции над стеком. Закрытые данные-члены будут обеспечивать хранение информации в стеке. Концепция класса хорошо соответствует подходу ADT.

Раздел `private` должен позаботиться о хранении данных. Например, можно использовать обычный массив, динамически распределенный в памяти массив либо какую-то более развитую структуру данных вроде связного списка. Однако открытый интерфейс класса должен скрывать точные детали представления. Наоборот, он должен быть выражен в общих понятиях, таких как создание стека, заталкивание элемента и т.д. В листинге 10.10 показан один из возможных подходов. Предполагается, что тип `bool` реализован. Если же это не так, то вместо `bool` с `false` и `true` можно использовать `int` со значениями `0` и `1`.

Листинг 10.10. `stack.h`

```
// stack.h -- определение класса для абстрактного типа данных — стека
#ifndef STACK_H_
#define STACK_H_

typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};        // константа, специфичная для класса
    Item items[MAX];        // хранит элементы стека
    int top;                // индекс вершины стека
public:
    Stack();
    bool isempty() const;
    bool isfull() const;
```

```
    // push() возвращает false, если стек полон, и true - в противном случае
    bool push(const Item & item);    // добавляет элемент в стек

    // pop() возвращает false, если стек пуст, и true - в противном случае
    bool pop(Item & item);           // выталкивает элемент с вершины стека
};
#endif
```

В примере, приведенном в листинге 10.10, раздел `private` показывает, что стек реализован с помощью массива, но раздел `public` никак не отражает этот факт. То есть обычный массив можно заменить, скажем, динамическим массивом, не меняя интерфейс класса. Это означает, что изменение реализации стека не требует внесения изменений в код программы, которая будет его использовать. Вы просто перекомпилируете код реализации стека и скомпонуете его с кодом программы.

Представленный интерфейс несколько избыточен, т.к. `pop()` и `push()` возвращают информацию о состоянии стека (пуст или полон) вместо того, чтобы иметь тип `void`. Это обеспечивает дополнительные возможности по управлению переполнением стека и его очисткой. Можно использовать `isempty()` и `isfull()` для проверки перед попытками модификации стека или с помощью возвращаемых значений `push()` и `pop()` определять, удалась ли соответствующая операция.

Вместо того чтобы определять стек в терминах некоторого конкретного типа, класс описывает его в терминах общего типа `Item`. В данном случае заголовочный файл использует `typedef` для указания, что `Item` является `unsigned long`. Если вы хотите создать стек для хранения элементов типа `double` или структур, измените `typedef`, а объявление класса и определения методов останутся прежними. Шаблоны классов (см. главу 14) предлагают более мощный метод изоляции типа хранимых данных от проектного решения класса.

Далее потребуется реализовать методы класса. В листинге 10.11 показан один из возможных вариантов.

Листинг 10.11. `stack.cpp`

```
// stack.cpp -- функции-члены класса Stack
#include "stack.h"
Stack::Stack()  // создание пустого стека
{
    top = 0;
}
bool Stack::isempty() const
{
    return top == 0;
}
bool Stack::isfull() const
{
    return top == MAX;
}
bool Stack::push(const Item & item)
{
    if (top < MAX)
    {
        items[top++] = item;
        return true;
    }
    else
        return false;
}
```

```
bool Stack::pop(Item & item)
{
   if (top > 0)
   {
      item = items[--top];
      return true;
   }
   else
      return false;
}
```

Конструктор по умолчанию гарантирует, что все стеки будут создаваться пустыми. Код функций-членов `pop()` и `push()` гарантирует корректное управление вершиной стека. Подобного рода гарантии — это одно из обстоятельств, которые делают ООП надежным. Предположим, что вы решили создать отдельный массив для представления стека и независимую переменную, представляющую индекс вершины стека. В этом случае вы отвечаете за правильность кода при каждом создании нового стека. Без защиты, предоставляемой закрытыми данными, всегда существует возможность сделать ошибку и изменить данные нежелательным образом.

Давайте протестируем стек. Код в листинге 10.12 моделирует деятельность клерка, обрабатывающего заказы на покупки, который берет их со стопки на столе, используя характерный для стека алгоритм LIFO (last-in, first-out — последним пришел, первым обслужен).

Листинг 10.12. `stacker.cpp`

```
// stacker.cpp -- тестирование класса Stack
#include <iostream>
#include <cctype>   // или ctype.h
#include "stack.h"
int main()
{
   using namespace std;
   Stack st;         // создание пустого стека
   char ch;
   unsigned long po;
   // A - добавление заказа, P - обработка заказа, Q - завершение
   cout << "Please enter A to add a purchase order,\n"
        << "P to process a PO, or Q to quit.\n";
   while (cin >> ch && toupper(ch) != 'Q')
   {
      while (cin.get() != '\n')
         continue;
      if (!isalpha(ch))
      {
         cout << '\a';
         continue;
      }
      switch(ch)
      {
         case 'A':
         case 'a': cout << "Enter a PO number to add: ";     // запрос номера заказа
                   cin >> po;
                   if (st.isfull())
                      cout << "stack already full\n";         // стек уже полон
                   else
```

```
                st.push(po);
                break;
        case 'P':
        case 'p': if (st.isempty())
                    cout << "stack already empty\n";        // стек уже пуст
                else {
                    st.pop(po);
                    cout << "PO #" << po << " popped\n";    // заказ вытолкнут
                }
                break;
    }
    cout << "Please enter A to add a purchase order,\n"
         << "P to process a PO, or Q to quit.\n";
  }
  cout << "Bye\n";
  return 0;
}
```

Небольшой цикл `while` в листинге 10.12, который избавляется от остатка строки, пока что не является совершенно необходимым, однако он пригодится в модифицированной версии программы, которая будет рассматриваться в главе 14. Ниже показан пример запуска программы:

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 17885
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #17885 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 17965
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: 18002
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #18002 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #17965 popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
stack already empty
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
Q
Bye
```

Резюме

В ООП основное внимание акцентируется на представлении данных. Первый шаг к решению проблем программирования с помощью объектно-ориентированного подхода заключается в описании данных в терминах их интерфейса с программой, указывающего, как их использовать. После этого должен быть спроектирован класс, который реализует такой интерфейс. Обычно закрытые данные-члены хранят информацию, в то время как открытые функции-члены, также называемые методами, предлагают единственный способ доступа к данным. Класс комбинирует данные и методы в единый модуль, а закрытый способ доступа обеспечивает сокрытие данных.

Обычно объявление класса разделяется на две части, как правило, сохраняемые в разных файлах. Объявление класса с методами, представленными с помощью прототипов функций, попадает в заголовочный файл. Исходный код, составляющий функции-члены, попадает в файл методов. Такой подход позволяет отделить описание интерфейса от деталей реализации.

В принципе для того, чтобы использовать класс, необходимо знать только его открытый интерфейс. Конечно же, можно просматривать реализацию (если только класс не поставляется в скомпилированном виде), однако программа не должна зависеть от деталей реализации класса, как и знать, что какое-то значение, например, хранится в виде `int`. До тех пор, пока программа и класс взаимодействуют только через методы, определенные в интерфейсе, вы вольны совершенствовать обе части независимо, не заботясь о нежелательном взаимодействии.

Класс — это определяемый пользователем тип, а объект — экземпляр класса. Это значит, что объект является переменной этого типа или эквивалентом переменной, такой как выделенный операцией `new` участок памяти в соответствии со спецификациями класса. C++ старается сделать применение пользовательских типов настолько же простым, как и стандартных типов, поэтому можно объявлять объекты, указатели на объекты и массивы объектов. Вы можете передавать объекты в виде аргументов, возвращать их в качестве значений из функций и присваивать один объект другому объекту того же типа. Если предоставлен метод конструктора, объекты могут быть инициализированы во время создания. Если предусмотрен деструктор, он будет вызван при уничтожении объектов.

Каждый объект содержит собственную копию набора данных из объявления класса, но все объекты совместно используют методы класса. Если `mr_object` — это имя определенного объекта, а `try_me()` — его функция-член, то вызывать эту функцию можно с помощью операции членства (точки), т.е. `mr_object.try_me()`. В терминологии ООП такой вызов называется отправкой сообщения `try_me()` объекту `mr_object`.

Любая ссылка на данные-члены класса в методе `try_me()` затем применяется к данным-членам объекта `mr_object`. Аналогичным образом, вызов функции `i_object.try_me()` получает доступ к данным-членам объекта `i_object`.

Если нужно, чтобы функция-член взаимодействовала с более чем одним объектом, ей следует передать дополнительные объекты в виде аргументов. Если метод нуждается в явном доступе к объекту, который его вызвал, он может сделать это через указатель `this`. Указатель `this` устанавливается в адрес вызывающего объекта, поэтому выражение `*this` является псевдонимом самого объекта.

Классы хорошо подходят для описания абстрактных типов данных (ADT). Интерфейс открытых функций-членов предоставляет службы, описанные ADT, а закрытые члены и код методов класса являются реализацией, скрытой от клиентов класса.

Вопросы для самоконтроля

1. Что такое класс?
2. Каким образом класс обеспечивает абстракцию, инкапсуляцию и сокрытие данных?
3. Каково отношение между объектом и классом?
4. Чем отличаются функции-члены класса от данных-членов класса помимо того, что они — функции?
5. Определите класс для представления банковского счета. Данные-члены должны включать имя вкладчика, номер счета (используйте строку) и баланс. Функции-члены должны позволять следующее:
   - создание объекта и его инициализация;
   - отображение имени вкладчика, номера счета и баланса;
   - добавление на счет суммы денег, переданной в аргументе;
   - снятие суммы денег, переданной в аргументе.

   Просто приведите объявление класса без реализации методов. (Возможность написать реализацию будет представлена в упражнении 1.)
6. Когда вызываются конструкторы класса? Когда вызываются деструкторы?
7. Напишите код конструктора для класса банковского счета, описанного в вопросе 5.
8. Что такое конструктор по умолчанию? Каковы выгоды его применения?
9. Модифицируйте определение класса `Stock` (версию в `stock20.h`) так, чтобы он имел функции-члены, которые возвращают значения индивидуальных данных-членов. На заметку: член, который возвращает наименование компании, не должен давать возможности изменять массив. То есть он не может просто возвращать ссылку на `string`. Он может возвращать `const`-ссылку.
10. Что такое `this` и `*this`?

Упражнения по программированию

1. Предоставьте определения методов для класса, описанного в вопросе 5, и напишите короткую программу для иллюстрации всех его возможностей.
2. Пусть имеется определение следующего простого класса:

```
class Person {
private:
   static const LIMIT = 25;
   string lname;              // фамилия
   char fname[LIMIT];         // имя
public:
   Person() { lname = ""; fname[0] = '\0'; }              // #1
   Person(const string & ln, const char * fn = "Heyyou"); // #2

   // Следующие методы отображают lname и fname
   void Show() const;         // формат: имя фамилия
   void FormalShow() const;   // формат: фамилия, имя
};
```

(В нем используется объект string и символьный массив, так что вы сможете сравнить применение этих двух форм.) Напишите программу, которая дополнит реализацию за счет предоставления кода для пока еще не определенных методов. В программе, использующей класс, должны также присутствовать вызовы трех возможных конструкторов (без аргументов, с одним аргументом, с двумя аргументами) и двух методов отображения. Ниже приведен пример примснения этих конструкторов и методов:

```
Person one;                          // используется конструктор по умолчанию
Person two("Smythecraft");           // используется конструктор #2
                                     // с одним аргументом по умолчанию
Person three("Dimwiddy", "Sam");     // используется конструктор #2,
                                     // без аргументов по умолчанию
one.Show();
cout << endl;
one.FormalShow();
// и т.д. для объектов two и three
```

3. Выполните упражнение 1 из главы 9, но замените показанный там код подходящим объявлением класса golf. Замените setgolf(golf &, const chart *, int) конструктором с соответствующими аргументами для выполнения инициализации. Оставьте интерактивную версию setgolf(), но реализуйте ее с использованием этого конструктора. (Например, в коде setgolf() получите данные, передайте их конструктору для создания временного объекта и присвойте временный объект вызвавшему, представленному через *this.)

4. Выполните упражнение 4 из главы 9, но преобразуйте структуру Sales и ассоциированные с ней функции в класс и методы. Замените функцию setSales(Sales &, double[], int) конструктором. Реализуйте интерактивный метод setSales(Sales &), используя конструктор. Оставьте класс в пространстве имен SALES.

5. Пусть имеется следующее объявление структуры:

```
struct customer {
   char fullname[35];
   double payment;
};
```

Напишите программу, которая будет добавлять структуры заказчиков в стек и удалять их из стека, представленного объявлением класса Stack. Всякий раз, когда заказчик удаляется из стека, его зарплата должна добавляться к промежуточной сумме и по этой сумме выдаваться отчет. На заметку: вы должны иметь возможность пользоваться классом Stack без изменений; просто поменяйте объявление typedef, чтобы Item был типом customer вместо unsigned long.

6. Пусть имеется следующее объявление класса:

```
class Move
{
private:
   double x;
   double y;
public:
   Move(double a = 0, double b = 0); // устанавливает x, y в a, b
   showmove() const;                 // отображает текущие значения x, y
   Move add(const Move & m) const;
```

```
    // Эта функция добавляет x из m к x вызывающего объекта,
    // чтобы получить новое значение x,
    // Добавляет y из m к y вызывающего объекта, чтобы получить новое
    // значение y, присваивает инициализированному объекту значения x, y
    // и возвращает его
    reset(double a = 0, double b = 0); // сбрасывает x, y в a, b
};
```

Создайте определения функций-членов и напишите программу, которая использует этот класс.

7. Плорг из Бетельгейзе обладает следующими свойствами:

   Данные

   - плорг имеет имя не длиннее 19 символов;
   - плорг имеет индекс удовлетворенности (contentment index — CI), выражаемый целым числом.

   Операции

   - новый плорг начинает существование с именем и индексом CI равным 50;
   - индекс CI плорга может изменяться;
   - плорг может сообщать свое имя и индекс CI;
   - по умолчанию плорг имеет имя `"Plorga"`.

   Напишите объявление класса `Plorg` (включая данные-члены и прототипы функций-членов), который представляет плорга. Напишите определения функций-членов. Напишите короткую программу, демонстрирующую все средства класса `Plorg`.

8. Простой список можно описать следующим образом:

   - простой список может содержать ноль или более элементов определенного типа;
   - можно создавать пустой список;
   - можно добавлять элемент в список;
   - можно определять, пуст ли список;
   - можно определять, полон ли список.
   - можно посетить каждый элемент списка и выполнить над ним определенное действие.

   Как видите, список действительно прост; так, например, он не позволяет осуществлять вставку или удаление элементов.

   Спроектируйте класс `List` для представления этого абстрактного типа. Вы должны подготовить заголовочный файл `list.h` с объявлением класса и файл `list.cpp` с реализацией его методов. Вы должны также написать короткую программу, которая будет использовать полученный класс.

   Главная причина того, что спецификация списка проста, связана с попыткой упростить это упражнение. Вы можете реализовать список в виде массива или же в виде связного списка, если знакомы с этим типом данных. Однако открытый интерфейс не должен зависеть от вашего выбора. То есть открытый интерфейс не должен иметь индексов массива, указателей на узлы и т.п. Он должен быть выражен в виде общих концепций создания списка, добавления элемента в список и т.д.

Обычный способ управления посещением каждого элемента в списке и выполнения над ним каких-то действий состоит в применении функции, которая принимает указатель на другую функцию в качестве аргумента:

```
void visit(void (*pf)(Item &));
```

Здесь `pf` указывает на функцию (не функцию-член), которая принимает ссылку на аргумент типа `Item`, где `Item` — это тип элементов списка. `visit()` применяет эту функцию к каждому элементу списка. В качестве общего руководства можете воспользоваться классом `Stack`.

11

Работа с классами

В ЭТОЙ ГЛАВЕ...

- Перегрузка операций
- Дружественные функции
- Перегрузка операции << для вывода
- Члены состояния
- Использование `rand()` для генерации случайных чисел
- Автоматические преобразования и приведения типов для классов
- Функции преобразования классов

Классы C++ — это богатые возможностями, сложные и мощные средства. В главе 10 вы приступили к исследованию объектно-ориентированного программирования, начав с определения и использования простого класса. Вы видели, что класс определяется как тип данных, предназначенный для представления объекта, и также посредством функций-членов — операций, которые могут выполняться над этими данными. Вы изучили также две специальных разновидности функций-членов — конструктор и деструктор, которые управляют процессом создания и уничтожения объектов, построенных на основе спецификаций класса. В настоящей главе продолжается объяснение свойств классов, при этом основное внимание уделяется не общим принципам, а приемам проектирования классов. Некоторые из рассматриваемых средств могут показаться простыми, другие же — достаточно сложными. Для лучшего понимания новых средств предлагаются соответствующие примеры, с которыми следует поэкспериментировать. Что случится, если в функции применить обычный аргумент вместо аргумента, переданного по ссылке? Что произойдет, если не обеспечить освобождение чего-либо в деструкторе? Не бойтесь совершать ошибки: обычно удается научиться большему, исправляя допущенные ошибки, чем делать что-либо корректно, но механически. (Однако не следует полагать, что обилие ошибок неизбежно приведет к глубоким знаниям предмета.) В конце концов, вы будете вознаграждены более полным пониманием того, как работает C++, и что с помощью этого языка программирования может сделать.

Глава начинается с описания механизма перегрузки операций, который позволит использовать стандартные операции C++, такие как = и +, с объектами классов. Затем рассматривается понятие "друзей" — механизма C++, который дает возможность функциям, не являющимся членами класса, получать доступ к закрытым данным. И, наконец, будет показано, как заставить C++ выполнять автоматические преобразования типов при работе с классами. После того, как вы ознакомитесь с настоящей главой и главой 12, вы достигнете полного понимания роли конструкторов и деструкторов в классах. Кроме того, вы узнаете о некоторых дополнительных стадиях, которые нужно пройти в процессе проектирования и разработки классов.

Одной из трудностей при изучении C++, по крайней мере, на данном этапе, является огромный объем информации, которую необходимо запомнить. И, конечно, нет резона рассчитывать, что удастся запомнить все, до тех пор, пока вы не приобретете достаточный опыт. В этом смысле изучение C++ подобно освоению перегруженного разнообразными средствами сложного текстового процессора или электронной таблицы. Ни одно из средств не является таким уж сложным, но на практике выясняется, что большинство людей действительно хорошо знают только те возможности, которыми они пользуются регулярно, например, средство поиска текста или выделение курсивом. Вы можете периодически вспоминать, что надо бы почитать где-нибудь, каким образом генерируются альтернативные символы или создаются таблицы, но эти знания не станут частью вашего арсенала до тех пор, пока вы не столкнетесь с ситуациями, в которых они будут востребованы часто. Возможно, лучший подход к усвоению материала данной главы — начать пользоваться хотя бы некоторыми из новых средств в повседневной практике программирования на C++. По мере накопления опыта, а вместе с ним — понимания и оценки пользы от новых возможностей, вы сможете постепенно добавлять в свой арсенал новые средства C++. Как говорил Бьярне Страуструп, создатель C++, на конференции профессиональных программистов: "Упрощайте язык для себя. Не считайте себя обязанными применять все средства языка, и уж тем более не пытайтесь использовать их все в первый же день".

Перегрузка операций

Давайте рассмотрим прием, с помощью которого операциям над объектами можно придать более симпатичный вид. *Перегрузка операций* — это пример полиморфизма C++. В главе 8 было показано, что C++ позволяет определять несколько функций с одинаковыми именами и разной сигнатурой (списками аргументов). Это называлось *перегрузкой функций* или *функциональным полиморфизмом*. Цель такой перегрузки — позволить использовать одно и то же имя функции для некоторой базовой операции, несмотря на то, что она применяется к данным разных типов. Перегрузка операций расширяет концепцию перегрузки на операции, позволяя трактовать их множеством способов. На самом деле многие операции C++ (как и C) уже перегружены.

Например, операция *, когда применяется к адресу, выдает значение, хранимое по этому адресу. Но использование * с двумя числовыми величинами означает их перемножение. Для решения, какое действие нужно выполнить в каждом конкретном случае, определяется количеством и типом операндов.

Язык C++ позволяет распространить перегрузку операций на пользовательские типы, разрешая, скажем, применять символ + для сложения двух объектов. Для выяснения, какое именно определение данной операции следует использовать, компилятор также использует количество и тип операндов. Перегруженные операции часто могут заставить код выглядеть более естественно. Например, общей вычислительной задачей является сложение двух массивов. Обычно это выглядит подобно следующему циклу `for`:

```
for (int i = 0; i < 20; i++)
   evening[i] = sam[i] + janet[i]; // поэлементное сложение
```

Но в C++ можно определить класс, который представляет массивы и перегружает операцию + таким образом, что станет возможным приведенный ниже код:

```
evening = sam + janet;              // сложить два объекта-массива
```

В приведенной простой нотации сложения скрывается внутренний механизм, но подчеркивается то, что существенно, а это и является одной из целей ООП.

Для перегрузки операции используется специальная форма функции, называемая *функцией операции*. Функция операции имеет следующую форму, в которой *op* — это символ перегружаемой операции:

```
operatorop(список-аргументов)
```

Например, `operator+()` перегружает операцию +, а `operator*()` — операцию *. Операция *op* должна быть допустимой операцией C++, а не произвольным символом. Например, объявить функцию `operator@()` не получится, т.к. в C++ нет операции @. С другой стороны, функция `operator[]()` перегружает операцию `[]`, поскольку `[]` — это операция индексации массивов. Предположим, что имеется класс `Salesperson`, в котором определена функция-член `operator+()` для перегрузки операции + так, что она сможет добавлять зарплату одного лица к зарплате другого лица. Тогда если `district2`, `sid` и `sara` — объекты класса `Salesperson`, можно написать следующее выражение:

```
district2 = sid + sara;
```

Компилятор, распознав операцию как относящуюся к классу `Salesperson`, заменит ее вызовом соответствующей функции операции:

```
district2 = sid.operator+(sara);
```

Функция затем использует объект sid неявно (поскольку она вызывает метод), а объект sara – явно (т.к. он передается в виде аргумента) для вычисления суммы, возвращаемой в результате. Конечно, возможность применять обозначение операции + вместо неуклюжего вызова функции выглядит более симпатично.

C++ накладывает некоторые ограничения на перегрузку операций, но все же их проще понять после того, как вы разберетесь, как работает перегрузка. Поэтому давайте создадим несколько примеров с целью прояснения процесса, после чего обсудим ограничения.

Время в наших руках: разработка примера перегрузки операции

Если вы находились в системе под конкретным именем пользователя в течение 2 часов 35 минут с утра и 2 часов 40 минут после обеда, то сколько всего времени вы проработали в системе? Ниже приведен пример, в котором концепция сложения имеет смысл, несмотря на то, что единицы, которые вы складываете (смесь часов и минут) не соответствует какому-либо встроенному типу. В главе 7 рассматривался подобный случай; там определялась структура travel_time и функция sum() для сложения структур упомянутого типа. Теперь давайте обобщим это в классе Time, используя метод для управления сложением. Начнем с простого метода, называемого Sum(), а затем посмотрим, как его преобразовать в перегруженную операцию. Объявление класса Time показано в листинге 11.1.

Листинг 11.1. mytime0.h

```
// mytime0.h -- класс Time до перегрузки операции
#ifndef MYTIME0_H_
#define MYTIME0_H_
class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time Sum(const Time & t) const;
    void Show() const;
};
#endif
```

Класс Time предоставляет методы для изменения и сброса времени, для отображения значений времени и для сложения двух значений времени. В листинге 11.2 приведено определение методов. Обратите внимание, что методы AddMin() и Sum() используют целочисленное деление и операцию взятия модуля для корректировки значений минут и часов, когда общее количество минут превышает 59. Также поскольку единственное средство iostream, которое здесь используется – это cout, а также потому, что оно применяется только один раз, имеет смысл указать std::cout вместо того, чтобы использовать все пространство имен std.

Листинг 11.2. mytime0.cpp

```
// mytime0.cpp -- реализация методов Time
#include <iostream>
#include "mytime0.h"
Time::Time()
{
    hours = minutes = 0;
}
Time::Time(int h, int m )
{
    hours = h;
    minutes = m;
}
void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr(int h)
{
    hours += h;
}
void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}
Time Time::Sum(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}
void Time::Show() const
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

Рассмотрим код функции `Sum()`. Обратите внимание, что аргумент является ссылкой, но возвращаемый тип — нет. Причина передачи аргумента по ссылке кроется в эффективности. Код будет давать те же самые результаты и при передаче объекта `Time` по значению, но с точки зрения использования памяти обычно быстрее и эффективнее передавать по ссылке.

Однако возвращаемое значение не может быть ссылкой. Это объясняется тем, что функция создает новый объект `Time` (по имени `sum`), который представляет сумму двух других объектов `Time`. Возврат объекта приводит к созданию копии объекта, которую может использовать вызывающая функция. Однако если возвращаемым типом является `Time &`, ссылка будет указывать на сам объект `sum`. Но объект `sum` — это локальная переменная, которая уничтожается при завершении работы функции, поэтому ссылка указывает на несуществующий объект. Использование типа возврата `Time` означает, что программа создает *копию* объекта `sum` перед его уничтожением, и вызывающая функция получает корректный результат.

Внимание!

Не возвращайте ссылку на локальную переменную или другой временный объект. Когда функция завершит работу и локальная переменная или временный объект исчезнут, ссылка станет указывать на несуществующие данные.

И, наконец, код в листинге 11.3 тестирует суммирование времени, реализовавшее классом `Time`.

Листинг 11.3. `usetime0.cpp`

```
// usetime0.cpp -- использование первой черновой версии класса Time
// компилировать usetime0.cpp и mytime0.cpp вместе
#include <iostream>
#include "mytime0.h"
int main()
{
   using std::cout;
   using std::endl;
   Time planning;
   Time coding(2, 40);
   Time fixing(5, 55);
   Time total;
   cout << "planning time = ";      // время на планирование
   planning.Show();
   cout << endl;
   cout << "coding time = ";        // время на кодирование
   coding.Show();
   cout << endl;
   cout << "fixing time = ";        // время на исправление
   fixing.Show();
   cout << endl;
   total = coding.Sum(fixing);
   cout << "coding.Sum(fixing) = ";
   total.Show();
   cout << endl;
   return 0;
}
```

Ниже показан вывод программы из листингов 11.1, 11.2 и 11.3.

```
planning time = 0 hours, 0 minutes
coding time = 2 hours, 40 minutes
fixing time = 5 hours, 55 minutes
coding.Sum(fixing) = 8 hours, 35 minutes
```

Добавление операции сложения

Класс `Time` очень просто изменить так, чтобы он использовал перегруженную операцию сложения. Понадобится только заменить имя функции `Sum()` выглядящим несколько странно именем `operator+()`. Однако здесь все правильно: необходимо добавить символ операции (в данном случае +) в конец слова `operator` и применить полученную строку в качестве имени метода. Это — единственное место, где в имени идентификатора допускается применение символа, отличного от букв, цифр и знака подчеркивания. Описанное небольшое изменение отражено в листингах 11.4 и 11.5.

Листинг 11.4. mytime1.h

```
// mytime1.h -- класс Time после перегрузки операции
#ifndef MYTIME1_H_
#define MYTIME1_H_
class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    void Show() const;
};
#endif
```

Листинг 11.5. mytime1.cpp

```
// mytime1.cpp -- реализация методов Time
#include <iostream>
#include "mytime1.h"
Time::Time()
{
    hours = minutes = 0;
}
Time::Time(int h, int m )
{
    hours = h;
    minutes = m;
}
void Time::AddMin(int m)
{
    minutes += m;
    hours += minutes / 60;
    minutes %= 60;
}
void Time::AddHr(int h)
{
    hours += h;
}
void Time::Reset(int h, int m)
{
    hours = h;
    minutes = m;
}
Time Time::operator+(const Time & t) const
{
    Time sum;
    sum.minutes = minutes + t.minutes;
    sum.hours = hours + t.hours + sum.minutes / 60;
    sum.minutes %= 60;
    return sum;
}
void Time::Show() const
{
    std::cout << hours << " hours, " << minutes << " minutes";
}
```

Подобно `Sum()`, метод `operator+()` вызывается объектом `Time`, принимая второй объект `Time` в качестве аргумента, а возвращает объект `Time`. Таким образом, метод `operator+()` можно вызвать с использованием того же синтаксиса, что и в случае с `Sum()`:

```
total = coding.operator+(fixing);       // нотация с функцией
```

Но назначение методу имени `operator+()` позволяет также применить нотацию операции:

```
total = coding + fixing;                // нотация с операцией
```

Оба варианта вызывают метод `operator+()`. Обратите внимание, что в нотации с операцией объект слева от операции (в рассматриваемом случае `coding`) является вызывающим объектом, а объект справа (в данном случае `fixing`) передается в качестве аргумента. Код в листинге 11.6 иллюстрирует это.

Листинг 11.6. `usetime1.cpp`

```
// usetime1.cpp -- использование второй черновой версии класса Time
// компилировать usetime1.cpp и mytime1.cpp вместе
#include <iostream>
#include "mytime1.h"

int main()
{
    using std::cout;
    using std::endl;
    Time planning;
    Time coding(2, 40);
    Time fixing(5, 55);
    Time total;

    cout << "planning time = ";      // время на планирование
    planning.Show();
    cout << endl;
    cout << "coding time = ";        // время на кодирование
    coding.Show();
    cout << endl;
    cout << "fixing time = ";        // время на исправление
    fixing.Show();
    cout << endl;
    total = coding + fixing;

    // Нотация с операцией
    cout << "coding + fixing = ";    // кодирование + исправление
    total.Show();
    cout << endl;
    Time morefixing(3, 28);
    cout << "more fixing time = ";   // дополнительное время на исправление
    morefixing.Show();
    cout << endl;
    total = morefixing.operator+(total);

    // Нотация с функцией
    cout << "morefixing.operator+(total) =";
    total.Show();
    cout << endl;
    return 0;
}
```

Ниже показан вывод программы из листингов 11.4, 11.5 и 11.6:

```
planning time = 0 hours, 0 minutes
coding time = 2 hours, 40 minutes
fixing time = 5 hours, 55 minutes
coding + fixing = 8 hours, 35 minutes
more fixing time = 3 hours, 28 minutes
morefixing.operator+(total) = 12 hours, 3 minutes
```

Короче говоря, имя функции `operator+()` позволяет вызывать ее как в нотации с функцией, так и в нотации с операцией. Компилятор использует тип операнда для определения того, что необходимо делать:

```
int a, b, c;
Time A, B, C;
c = a + b;    // используется сложение значений int
C = A + B;    // используется сложение, определенное для объектов Time
```

А можно ли складывать более двух объектов? Например, если `t1`, `t2`, `t3` и `t4` являются объектами класса `Time`, будет ли допустимым следующий оператор:

```
t4 = t1 + t2 + t3;                    // правильно ли это?
```

Чтобы ответить на этот вопрос, нужно посмотреть, как это выражение транслируется в вызовы функций. Поскольку сложение — операция, выполняемая слева направо, первая трансляция дает:

```
t4 = t1.operator+(t2 + t3);           // правильно ли это?
```

Затем аргумент функции также транслируется в вызов функции, и мы получаем:

```
t4 = t1.operator+(t2.operator+(t3));    // правильно ли это? ДА
```

Верно ли это? Да, верно. Вызов функции `t2.operator+(t3)` возвращает объект `Time`, представляющий собой сумму `t2` и `t3`. Этот объект затем передается вызову `t1.operator+()` и этот вызов возвращает сумму `t1` и объекта `Time`, который представляет сумму `t2` и `t3`. Короче говоря, финальное возвращаемое значение является суммой `t1`, `t2` и `t3`, что и ожидалось.

Ограничения перегрузки

Большинство операций C++ (описанные в табл. 11.1) могут быть перегружены так, как было описано выше. Перегруженные операции (за некоторыми исключениями) не обязательно должны быть функциями-членами. Однако, по крайней мере, один из операндов должен иметь тип, определяемый пользователем. Давайте посмотрим внимательнее на ограничения, которые накладывает C++ на перегрузку операций, определяемых пользователем.

- Перегруженные операции должны иметь как минимум один операнд типа, определяемого пользователем. Это предотвращает перегрузку операций, работающих со стандартными типами. То есть переопределить операцию "минус" (-) так, чтобы она вычисляла сумму двух вещественных чисел вместо разности, не получится. Это ограничение сохраняет здравый смысл, заложенный в программу, хотя и несколько препятствует полету творчества.
- Вы не можете использовать операцию в такой манере, которая нарушает правила синтаксиса исходной операции.

Например, нельзя перегрузить операцию взятия модуля (%) так, чтобы она применялась с одним операндом:

```
int x;
Time shiva;
% x;         // не допускается для операции взятия модуля
% shiva;     // не допускается для перегруженной операции
```

Аналогично, не допускается изменение приоритетов операций. Поэтому, если вы перегрузите операцию сложения для класса, то новая операция будет иметь тот же приоритет, что и обычное сложение.

- Вы не можете определять новые символы операций. Например, определить функцию `operator**()` для создания операции возведения в степень не получится.
- Нельзя перегружать следующие операции:

Операция	Описание
`sizeof`	Операция `sizeof`
`.`	Операция членства
`.*`	Операция указателя на член
`::`	Операция разрешения контекста
`:?`	Условная операция
`typeid`	Операция RTTI (runtime type identification — определение типа во время выполнения)
`const_cast`	Операция приведения типа
`dynamic_cast`	Операция приведения типа
`reinterpret_cast`	Операция приведения типа
`static_cast`	Операция приведения типа

Тем не менее, операции, перечисленные в табл. 11.1, по-прежнему доступны для перегрузки.

- Большинство операций из табл. 11.1 допускают перегрузку за счет использования как функций-членов, так и функций, не являющихся членами. Однако для перегрузки перечисленных ниже операций можно использовать *только* функции-члены:

Операция	Описание
`=`	Операция присваивания
`()`	Операция вызова функции
`[]`	Операция индексации
`->`	Операция доступа к членам класса через указатель

На заметку!

В настоящей главе не раскрыты все операции, упомянутые в списке ограничений или в табл. 11.1. Операции, которые не рассмотрены в тексте этой главы, кратко описаны в приложении Д.

Таблица 11.1. Операции, которые могут быть перегружены

+	-	*	/	%	^
&	|	~	!	=	<
>	+=	-=	*=	/=	%=
^=	&=	|=	<<	>>	>>=
<<=	==	!=	<=	>=	&&
||	++	--	,	->*	->
()	[]	new	delete	new []	delete []

В дополнение к этим формальным ограничениям при перегрузке операций вы должны использовать смысловые ограничения. Например, вы не должны перегружать операцию * так, что она будет обменивать значения данных-членов двух объектов `Time`. Если вы сделаете подобное, то в нотации ничего не будет указывать на то, *что* на самом деле операция делает, поэтому для этой цели лучше все-таки предусмотреть метод класса с осмысленным именем, например, `Swap()`.

Дополнительные перегруженные операции

Для класса `Time` имеют смысл и другие операции. Например, может понадобиться вычитать одно время из другого или умножать время на число. Представим, скажем, перегрузку операций вычитания и умножения. Подход здесь тот же, что и для операции сложения: создание методов `operator-()` и `operator*()`. То есть к объявлению класса добавляются следующие прототипы:

```
Time operator-(const Time & t) const;
Time operator*(double n) const;
```

В листинге 11.7 показан новый заголовочный файл.

Листинг 11.7. `mytime2.h`

```
// mytime2.h -- класс Time после перегрузки операции
#ifndef MYTIME2_H_
#define MYTIME2_H_
class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    Time operator-(const Time & t) const;
    Time operator*(double n) const;
    void Show() const;
};
#endif
```

После этого определения новых методов добавляются в файл реализации, приведенный в листинге 11.8.

Листинг 11.8. mytime2.cpp

```
// mytime2.cpp -- реализация методов Time
#include <iostream>
#include "mytime2.h"
Time::Time()
{
   hours = minutes = 0;
}
Time::Time(int h, int m )
{
   hours = h;
   minutes = m;
}
void Time::AddMin(int m)
{
   minutes += m;
   hours += minutes / 60;
   minutes %= 60;
}
void Time::AddHr(int h)
{
   hours += h;
}
void Time::Reset(int h, int m)
{
   hours = h;
   minutes = m;
}
Time Time::operator+(const Time & t) const
{
   Time sum;
   sum.minutes = minutes + t.minutes;
   sum.hours = hours + t.hours + sum.minutes / 60;
   sum.minutes %= 60;
   return sum;
}
Time Time::operator-(const Time & t) const
{
   Time diff;
   int tot1, tot2;
   tot1 = t.minutes + 60 * t.hours;
   tot2 = minutes + 60 * hours;
   diff.minutes = (tot2 – tot1) % 60;
   diff.hours = (tot2 – tot1) / 60;
   return diff;
}
Time Time::operator*(double mult) const
{
   Time result;
   long totalminutes = hours * mult * 60 + minutes * mult;
   result.hours = totalminutes / 60;
   result.minutes = totalminutes % 60;
   return result;
}
void Time::Show() const
{
   std::cout << hours << " hours, " << minutes << " minutes";
}
```

Наконец, измененные определения можно протестировать с помощью кода, показанного в листинге 11.9.

Листинг 11.9. usetime2.cpp

```
// usetime2.cpp -- использование третьей черновой версии класса Time
// Компилировать usetime2.cpp и mytime2.cpp вместе
#include <iostream>
#include "mytime2.h"
int main()
{
    using std::cout;
    using std::endl;
    Time weeding(4, 35);
    Time waxing(2, 47);
    Time total;
    Time diff;
    Time adjusted;

    cout << "weeding time = ";       // время на подготовку
    weeding.Show();
    cout << endl;

    cout << "waxing time = ";        // полезное время
    waxing.Show();
    cout << endl;

    cout << "total work time = ";    // общее рабочее время
    total = weeding + waxing;        // используется operator+()
    total.Show();
    cout << endl;

    diff = weeding - waxing;         // используется operator-()
    cout << "weeding time - waxing time = ";
    diff.Show();
    cout << endl;

    adjusted = total * 1.5;          // используется operator+()
    cout << "adjusted work time = ";
    adjusted.Show();
    cout << endl;

    return 0;
}
```

Вот как выглядит вывод программы из листингов 11.7, 11.8 и 11.9:

```
weeding time = 4 hours, 35 minutes
waxing time = 2 hours, 47 minutes
total work time = 7 hours, 22 minutes
weeding time - waxing time = 1 hours, 48 minutes
adjusted work time = 11 hours, 3 minutes
```

Что такое друзья?

Как вы уже видели, C++ управляет доступом к разделу `private` объекта класса. Обычно открытые (`public`) методы класса служат единственным каналом доступа, но иногда такое ограничение оказывается чересчур строгим, чтобы удовлетворять некоторым потребностям, возникающим в процессе программирования. Для таких случаев в C++ предусмотрена другая форма доступа — *друзья*.

Существует три разновидности друзей:

- дружественные функции;
- дружественные классы;
- дружественные функции-члены.

Объявляя функцию другом класса, вы позволяете ей иметь те же привилегии доступа, что и у функций-членов класса. Дружественные функции подробно рассматриваются в этой главе, а остальные две разновидности будут объясняться в главе 15.

Прежде чем мы увидим, как определяются друзья, давайте посмотрим, зачем они вообще могут понадобиться. Часто перегрузка бинарной операции (т.е. операции с двумя аргументами) приводит к потребности в друзьях. Умножение объекта `Time` на вещественное число как раз представляет такую ситуацию, поэтому давайте исследуем ее.

В предшествующем примере класса `Time` перегруженная операция умножения отличается от двух других перегруженных операций тем, что комбинирует два разных типа. То есть операции сложения и вычитания работают с двумя значениями типа `Time`, а операция умножения комбинирует значение типа `Time` со значением типа `double`. Это ограничивает ее применение. Помните, что левый операнд – это вызывающий объект. То есть

```
A = B * 2.75;
```

транслируется в следующий вызов функции-члена:

```
A = B.operator*(2.75);
```

Но как насчет приведенного ниже оператора?

```
A = 2.75 * B;    // не соответствует функции-члену
```

Концептуально `2.75 * B` должно быть эквивалентно `B * 2.75`, но первое выражение не может соответствовать функции-члену, поскольку `2.75` не является объектом типа `Time`. Помните, что левый операнд – это вызывающий объект, но `2.75` – не объект. Поэтому компилятор не может заменить это выражение вызовом функции-члена.

Один из способов обойти эту трудность – сказать всем (и запомнить самому), что допускается только запись в виде `B * 2.75`, но не `2.75 * B`. Это дружественное к серверу решение, возлагающее ответственность на клиента, что не отвечает принципам ООП.

Однако существует другая возможность – использовать функцию, не являющуюся членом. (Вспомните, что большинство операций могут быть перегружены с применением как функций-членов, так и просто функций.) Функция, не являющаяся членом, не вызывается через объект. Вместо этого все значения, которые она использует, включая объекты, передаются в виде явных аргументов. Таким образом, компилятор может представить выражение

```
A = 2.75 * B;    // не соответствует функции-члену
```

в виде вызова функции, не являющейся членом:

```
A = operator*(2.75, B);
```

Эта функция должна иметь следующий прототип:

```
Time operator*(double m, const Time & t);
```

Благодаря этой функции, не являющейся членом, которая перегружает операцию, левый операнд выражения соответствует первому аргументу функции, а правый – второму. Между тем, исходная функция-член имеет дело с операндами в обратном порядке – т.е. значение типа `Time` умножается на значение типа `double`.

Применение функции, не являющейся членом, решает проблему получения операндов в нужном порядке (сначала double, затем — Time), но приводит к возникновению новой проблемы: такая функция не имеет непосредственного доступа к закрытым данным класса. Во всяком случае, доступа не имеет обычная функция, не являющаяся членом. Однако существует специальная категория функций, не являющихся членами, называемая *друзьями*, которая имеет доступ к закрытым данным класса.

Создание друзей

Первый шаг в создании дружественной функции предусматривает помещение прототипа в объявление класса и предварение его префиксом в виде ключевого слова friend:

```
friend Time operator*(double m, const Time & t); // размещается в объявлении класса
```

На основе этого прототипа можно сделать два вывода.

- Несмотря на то что функция operator*() присутствует в объявлении класса, она не является функцией-членом класса. Поэтому она не вызывается через операцию членства (.).
- Несмотря на то что функция operator*() не является функцией-членом класса, она имеет те же права доступа, что и функции-члены.

Второй шаг состоит в написании определения функции. Поскольку она не является функцией-членом, добавлять квалификатор Time:: не нужно. Кроме того, ключевое слово friend также не используется в определении. Определение будет выглядеть следующим образом:

```
Time operator*(double m, const Time & t) // friend в определении не используется
{
   Time result;
   long totalminutes = t.hours * mult * 60 +t. minutes * mult;
   result.hours = totalminutes / 60;
   result.minutes = totalminutes % 60;
   return result;
}
```

С таким объявлением и определением оператор

```
A = 2.75 * B;
```

транслируется в следующий вызов только что определенной дружественной функции, не являющейся членом:

```
A = operator*(2.75, B);
```

Короче говоря, дружественная функция класса — это функция, не являющаяся членом, которая имеет те же права доступа, что и функция-член.

Нарушают ли друзья принципы ООП?

На первый взгляд может показаться, что друзья нарушают принцип сокрытия данных ООП, поскольку механизм друзей позволяет функциям, не являющимся членами, получать доступ к закрытым данным. Однако так может показаться только при поверхностном взгляде. Вместо этого вы должны думать о дружественных функциях, как о части расширенного интерфейса класса. Например, с концептуальной точки зрения умножение значения типа double на объект Time — то же самое, что и умножение объекта Time на значение типа double.

И хотя для реализации первого необходима дружественная функция, а для второго — функция-член, разница между ними выражается только в синтаксисе C++, а не в глубоком концептуальном смысле. Используя и дружественную функцию, и функцию-член, обе операции можно выразить в одном пользовательском интерфейсе. Кроме того, следует помнить, что только объявление класса определяет, какие функции являются дружественными, т.е. объявление класса по-прежнему управляет тем, каким функциям разрешен доступ к закрытым данным. Короче говоря, методы класса и друзья — это просто два разных механизма выражения интерфейса класса.

В действительности эту конкретную дружественную функцию можно реализовать через вызов функции-члена, изменив ее определение так, чтобы перед умножением она сначала меняла местами переданные операнды:

```
Time operator*(double m, const Time & t)
{
    return t * m;   // используется t.operator*(m)
}
```

Исходная версия имеет явный доступ к `t.minutes` и `t.hours`, поэтому должна быть другом. Показанная выше версия только использует объект `t` типа `Time` как единое целое, позволяя функции-члену работать с закрытыми значениями, поэтому такая функция не обязана быть другом. Однако, несмотря на это, все же имеет смысл сделать эту версию также дружественной. Самое главное, что это делает ее частью официального интерфейса класса. И второе: если вы позже столкнетесь с необходимостью иметь прямой доступ к закрытым данным, то должны будете изменить только определение функции, не затрагивая прототип класса.

Совет

Если вы хотите перегрузить операцию для класса и применять ее с первым операндом, не являющимся объектом класса, можете использовать дружественную функцию для изменения порядка следования операндов.

Общий вид друга: перегрузка операции <<

Очень удобным свойством классов является возможность перегрузить операцию << так, чтобы использовать ее с `cout` для отображения содержимого объекта. В некотором роде такая перегрузка немного сложнее, чем в предыдущих примерах, поэтому она будет разработана за два шага, а не один.

Предположим, что `trip` — это объект типа `Time`. Для отображения значений `Time` используется метод `Show()`. Однако разве не было бы лучше, если бы можно было записать так:

```
cout << trip;      // научить cout распознавать класс Time?
```

Это можно сделать, поскольку << — одна из операций C++, допускающих перегрузку. Фактически, она уже в большой степени перегружена. В своей базовой реализации операция << является одной из операций C и C++, предназначенных для манипулирования битами; она сдвигает биты значения на один влево (см. приложение Д). Но класс `ostream` перегружает эту операцию, превращая ее в инструмент вывода. Вспомните, что `cout` — объект типа `ostream`, и он достаточно интеллектуален, чтобы распознавать все базовые типы C++. Это потому, что объявление класса `ostream` включает перегруженное определение `operator<<()` для каждого из базовых типов. То есть одно определение использует аргумент типа `int`, другое — `double` и т.д. Поэтому одним из способов научить `cout` распознавать объекты `Time` является добавление нового опре-

деления функции операции к объявлению класса ostream. Однако изменение заголовочного файла iostream и внесение путаницы в стандартный интерфейс — опасная идея. Гораздо лучше научить класс Time использованию cout.

Первая версия перегрузки операции <<

Чтобы научить класс Time взаимодействовать с cout, можно воспользоваться дружественной функцией. Почему? Да потому, что оператор, подобный показанному ниже, работает с двумя объектами, причем объект класса ostream (а именно — cout) идет первым:

```
cout << trip;
```

Если воспользоваться функцией-членом Time для перегрузки <<, то объект Time должен следовать первым, как это было в примере с перегрузкой операции * с помощью функции-члена. Это означает, что операцию << пришлось бы применять следующим образом:

```
trip << cout;  // если бы operator<<() была функцией-членом Time
```

Такая запись вполне может сбить с толку. Но за счет использования дружественной функции операцию можно перегрузить, как показано ниже:

```
void operator<<(ostream & os, const Time & t)
{
   os << t.hours << " hours, " << t.minutes << " minutes";
}
```

Это позволит написать следующий код:

```
cout << trip;
```

который отобразит данные в таком формате:

```
4 hours, 23 minutes
```

Друг или не друг?

В новом объявлении класса Time функция operator<<() сделана дружественной функцией класса Time. Однако эта функция, хотя и не "враждебна" классу ostream, дружественной по отношению к нему не является. Функция operator<<() принимает аргументы ostream и Time, поэтому может показаться, что она должна быть дружественной по отношению к обоим классам. Но если вы посмотрите на код этой функции, то увидите, что она получает доступ к индивидуальным членам объекта Time, но использует объект ostream только как единое целое. Поскольку operator<<() обращается к закрытым членам Time напрямую, она должна быть другом класса Time. Так как она не имеет доступа к закрытым членам класса ostream, то другом этого класса ей быть не нужно. И это замечательно, поскольку означает, что изменять определение ostream не понадобится.

Обратите внимание, что новое определение operator<<() принимает ссылку на объект os типа ostream в качестве первого аргумента. Обычно os является ссылкой на объект cout, как это имеет место в выражении cout << trip. Однако эту операцию можно использовать с другими объектами ostream; в этом случае os будет ссылаться на них.

Как? Вы не знаете других объектов ostream?

Другим объектом ostream является cerr, который направляет вывод в стандартный поток сообщений об ошибках (по умолчанию дисплей). Однако в Unix, Linux и в режиме командной строки Windows стандартный поток ошибок можно перенаправить в файл. Также вспомните, что в главе 6 были представлены объекты ofstream, которые можно использовать для отправки вывода в файл.

С помощью наследования (см. главу 13) объекты ofstream могут пользоваться методами ostream. Таким образом, определение operator<<() можно применить для вывода данных Time в файлы — точно так же, как они выводятся на экран. Необходимо просто передать в качестве первого аргумента соответствующим образом инициализированный объект ofstream вместо cout.

Вызов cout << trip должен использовать сам объект cout, а не его копию, поэтому функция передает данный объект по ссылке, а не по значению. То есть выражение cout << trip делает os псевдонимом cout, а выражение cerr << trip устанавливает os в качестве псевдонима cerr. Объект Time может быть передан по значению или по ссылке, т.к. обе формы делают его значения доступными для функции операции. Опять-таки, передача по ссылке требует меньших затрат памяти и времени, чем передача по значению.

Вторая версия перегрузки операции <<

С представленной выше реализацией связана одна проблема. Операторы вроде показанного ниже работают хорошо:

```
cout << trip;
```

Но данная реализация не позволяет применять переопределенную операцию << так, как это обычно делается при работе с cout:

```
cout << "Trip time: " << trip << " (Tuesday)\n";  // так не получится
```

Чтобы понять, почему это не работает, и какие действия следует предпринять для обеспечения его работоспособности, сначала нужно узнать немного больше о том, как обращаться с cout. Предположим, что есть следующие операторы:

```
int x = 5;
int y = 8;
cout << x << y;
```

C++ читает выражение вывода слева направо, подразумевая следующий эквивалент:

```
(cout << x) << y;
```

Операция <<, как она определена в iostream, принимает в левой части объект ostream. Ясно, что выражение cout << x удовлетворяет этому требованию, поскольку cout представляет собой объект ostream. Но оператор вывода также требует, чтобы все выражение (cout << x) было типа ostream, поскольку оно расположено слева от << y. Следовательно, класс ostream реализует функцию operator<<() так, что она возвращает ссылку на объект ostream. В частности, в данном случае она возвращает ссылку на вызывающий объект cout. Таким образом, выражение (cout << x) само по себе является объектом cout типа ostream и может находиться в левой части операции <<.

Тот же самый подход можно применить с дружественной функцией. Понадобится лишь изменить функцию operator<<() так, чтобы она возвращала ссылку на объект ostream:

```
ostream & operator<<(ostream & os, const Time & t)
{
    os << t.hours << " hours, " << t.minutes << " minutes";
    return os;
}
```

Обратите внимание, что типом возврата является `ostream &`. Вспомните, что это означает возврат функцией ссылки на объект `ostream`. Поскольку программа передает ссылку на объект функции в первом аргументе, общий эффект состоит в том, что функция возвращает тот самый объект, который ей передан. То есть оператор

```
cout << trip;
```

превращается в следующий вызов функции:

```
operator<<(cout, trip);
```

И такой вызов возвращает объект `cout`. Поэтому теперь работает такой оператор:

```
cout << "Trip time: " << trip << " (Tuesday)\n";  // теперь работает
```

Давайте разобьем его на отдельные шаги, чтобы увидеть, как он работает. Первый шаг вызывает конкретное определение операции << из `ostream`, которое отображает строку и возвращает объект `cout`:

```
cout << " Trip time: "
```

Поэтому выражение `cout << " Trip time: "` отображает строку и затем заменяется типом возврата — `cout`. Это превращает исходный оператор в следующий:

```
cout << trip << " (Tuesday)\n";
```

Далее программа использует определение операции << из класса `Time` для того, чтобы отобразить значения `trip` и снова возвратить объект `cout`. Оператор приобретет такой вид:

```
cout << " (Tuesday)\n";
```

Программа завершается использованием определения операции << из `ostream` для строк, чтобы отобразить результирующую строку.

Интересно, что эта версия `operator<<()` также может быть применена для вывода в файл:

```
#include <fstream>
...
ofstream fout;
fout.open("savetime.txt");
Time trip(12, 40);
fout << trip;
```

Последний оператор становится таким:

```
operator<<(fout, trip);
```

И, как показано в главе 8, механизм наследования классов позволяет ссылке на `ostream` обращаться и к объектам `ostream`, и к объектам `ofstream`.

Совет

В общем случае для перезагрузки операции << с целью отображения объекта класса *c_name* используется дружественная функция со следующим определением:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
   os << ... ;  // отображение содержимого объекта
   return os;
}
```

В листинге 11.10 показано модифицированное определение класса с включением двух дружественных функций `operator*()` и `operator<<()`. Первая из этих функций реализована здесь как встроенная, поскольку ее код очень короткий. (Когда определение одновременно является прототипом, как в этом случае, применяется префикс `friend`.)

> **На заметку!**
>
> Ключевое слово `friend` используется только в прототипе, представленном в объявлении класса. В определении функции оно указывается, только если не присутствует в самом прототипе.

Листинг 11.10. `mytime3.h`

```
// mytime3.h -- класс Time с друзьями
#ifndef MYTIME3_H_
#define MYTIME3_H_
#include <iostream>
class Time
{
private:
    int hours;
    int minutes;
public:
    Time();
    Time(int h, int m = 0);
    void AddMin(int m);
    void AddHr(int h);
    void Reset(int h = 0, int m = 0);
    Time operator+(const Time & t) const;
    Time operator-(const Time & t) const;
    Time operator*(double n) const;
    friend Time operator*(double m, const Time & t)
        { return t * m; }  // встроенное определение
    friend std::ostream & operator<<(std::ostream & os, const Time & t);
};
#endif
```

В листинге 11.11 показан пересмотренный набор определений. Снова обратите внимание на то, что методы используют квалификатор `Time::`, тогда как дружественные функции — нет. Кроме того, поскольку `mytime3.h` включает `iostream` и предоставляет объявление `using std::ostream`, включение файла `mytime3.h` в `mytime3.cpp` предоставляет поддержку `ostream` в файле реализации.

Листинг 11.11. `mytime3.cpp`

```
// mytime3.cpp — реализация методов класса Time
#include "mytime3.h"
Time::Time()
{
    hours = minutes = 0;
}

Time::Time(int h, int m )
{
    hours = h;
    minutes = m;
}
```

```
void Time::AddMin(int m)
{
   minutes += m;
   hours += minutes / 60;
   minutes %= 60;
}

void Time::AddHr(int h)
{
   hours += h;
}

void Time::Reset(int h, int m)
{
   hours = h;
   minutes = m;
}

Time Time::operator+(const Time & t) const
{
   Time sum;
   sum.minutes = minutes + t.minutes;
   sum.hours = hours + t.hours + sum.minutes / 60;
   sum.minutes %= 60;
   return sum;
}

Time Time::operator-(const Time & t) const
{
   Time diff;
   int tot1, tot2;
   tot1 = t.minutes + 60 * t.hours;
   tot2 = minutes + 60 * hours;
   diff.minutes = (tot2 - tot1) % 60;
   diff.hours = (tot2 - tot1) / 60;
   return diff;
}

Time Time::operator*(double mult) const
{
   Time result;
   long totalminutes = hours * mult * 60 + minutes * mult;
   result.hours = totalminutes / 60;
   result.minutes = totalminutes % 60;
   return result;
}

std::ostream & operator<<(std::ostream & os, const Time & t)
{
   os << t.hours << " hours, " << t.minutes << " minutes";
   return os;
}
```

В листинге 11.12 приведен код программы-примера.

Формально файл `usetime3.cpp` не должен включать заголовочный файл `iostream`, поскольку `mytime3.h` уже включает его. Однако, как пользователь класса `Time`, вы не обязаны знать, какие заголовочные файлы включены в код класса, поэтому в вашей ответственности включать эти файлы, если код в них нуждается.

Листинг 11.12. usetime3.cpp

```
// usetime3.cpp -- использование четвертой черновой версии класса Time
// Компилировать usetime3.cpp и mytime3.cpp вместе
#include <iostream>
#include "mytime3.h"
int main()
{
    using std::cout;
    using std::endl;

    Time aida(3, 35);
    Time tosca(2, 48);
    Time temp;
    cout << "Aida and Tosca:\n";
    cout << aida<<"; " << tosca << endl;
    temp = aida + tosca;                          // operator+()
    cout << "Aida + Tosca: " << temp << endl;
    temp = aida * 1.17;                           // функция-член operator*()
    cout << "Aida * 1.17: " << temp << endl;
    cout << "10.0 * Tosca: " << 10.0 * tosca << endl;
    return 0;
}
```

Ниже показан вывод программы из листингов 11.10, 11.11 и 11.12:

```
Aida and Tosca:
3 hours, 35 minutes; 2 hours, 48 minutes
Aida + Tosca: 6 hours, 23 minutes
Aida * 1.17: 4 hours, 11 minutes
10.0 * Tosca: 28 hours, 0 minutes
```

Перегруженные операции: сравнение функций-членов и функций, не являющихся членами

При реализации перегрузки многих операций имеется выбор между функциями-членами и функциями, не являющимися членами. Как правило, версия, не являющаяся членом — это дружественная функция, в связи с чем она имеет доступ к закрытым данным класса. Например, рассмотрим операцию сложения класса `Time`. В объявлении класса `Time` она имеет следующий прототип:

```
// Вариант с функцией-членом
Time operator+(const Time & t) const;
```

Вместо этого класс может использовать такой прототип:

```
// Вариант с функцией, не являющейся членом
friend Time operator+(const Time & t1, const Time & t2);
```

Операция сложения требует два операнда. В версии с функцией-членом один операнд передается неявно, через указатель `this`, а второй — явно, как аргумент функции. В версии с дружественной функцией оба параметра передаются в качестве аргументов функции.

На заметку!

Версия перегруженной операции с функцией, не являющейся членом, требует столько формальных параметров, сколько операндов есть у данной операции. Версия с использованием функции-члена требует на один параметр меньше, поскольку один операнд передается неявно как вызывающий объект.

Оба эти прототипы соответствуют выражению `T2 + T3`, где `T2` и `T3` — объекты типа `Time`. То есть компилятор может преобразовать оператор

```
T1 = T2 + T3;
```

в один из следующих:

```
T1 = T2.operator+(T3);    // функция-член
T1 = operator+(T2, T3);   // функция, не являющаяся членом
```

Помните, что при определении этой операции вы должны выбрать одну или другую форму, но не обе сразу. Поскольку обеим формам соответствует одно и то же выражение, определение обеих форм одновременно ведет к неоднозначности и ошибке компиляции.

Итак, какую же форму стоит выбрать? Как упоминалось ранее, для некоторых операций единственно правильным выбором будет функция-член. В других случаях особой разницы между ними нет. Иногда, в зависимости от проектного решения, положенного в основу класса, вариант с использованием функции, не являющейся членом, может быть предпочтительнее, в частности, если для класса определены преобразования типов. Эта ситуация будет подробно обсуждаться в разделе "Преобразования и друзья" в конце настоящей главы.

Дополнительные сведения о перегрузке: класс `Vector`

Давайте рассмотрим другой класс, в котором используется перегрузка операций и друзья — класс, представляющий векторы. Этот класс также иллюстрирует дополнительные аспекты проектного решения, такие как внедрение в объект двух различных способов описания одной и той же вещи. Даже если векторы сейчас не интересуют, многие их приведенных здесь приемов можно использовать в другом контексте. *Вектор*, как он определяется в инженерной практике и физике — это величина, которая имеет модуль (размер) и направление. Например, если вы толкаете что-нибудь, то эффект от этого действия зависит от того, насколько сильно вы толкаете (модуль), и в каком направлении. Толчок в одном направлении может удержать шатающуюся вазу, а в другом — подтолкнуть ее к падению.

Чтобы полностью описать движение автомобиля, потребуется указать как его скорость (модуль), так и направление; если вы докажете патрульно-постовой службе, что не превышали скорости, то ваш аргумент немного будет стоить, если вы ехали против разрешенного направления движения. (Специалисты в иммунологии и в области вычислительной техники могут использовать термин *вектор* по-разному; отложим эти соображения, по крайней мере, до главы 16, где будет рассматриваться версия из вычислительной техники — шаблонный класс `vector`.) В следующей врезке даются дополнительные сведения о векторах, но полное их понимание не обязательно для отслеживания аспектов C++ в примерах.

Векторы

Представьте, что вы — рабочая пчела и открыли чудесный источник нектара. Вы возвращаетесь в улей и объявляете, что нашли нектар в 100 метрах от него. "Недостаточно информации" — жужжат остальные пчелы. "Надо сообщить нам и направление!". Ваш ответ: "30 градусов к северу от направления на солнце". Зная расстояние (модуль) и направление, другие пчелы смогут достичь этого сладкого места. Пчелы знакомы с понятием вектора.

Многие вещи описываются модулем и направлением. Например, эффект от толчка зависит как от его силы, так и направления. Перемещение объекта по экрану компьютера описывается расстоянием и направлением. Вещи подобного рода могут быть описаны с помощью векторов. Например, перемещение объекта по экрану можно описать вектором, который визуализируется в виде стрелки, соединяющей исходное положение с конечным. Длина вектора — это его модуль, описывающий, насколько далеко должна быть перемещена точка. Ориентация стрелки описывает направление (рис. 11.1). Вектор, представляющий такое изменение позиции, называется *вектором смещения*.

Теперь представьте, что вы — Лханаппа, великий охотник на мамонтов. Разведчики сообщают, что видели мамонта в 14,1 километрах к северо-западу. Но поскольку дует юго-восточный ветер, вы не можете приближаться к нему с юго-востока. Поэтому вы перемещаетесь на 10 километров к западу, а затем на 10 километров к северу, приближаясь к нему с юга. Вы знаете, что два этих вектора смещения приведут вас в ту же точку, как и один вектор в 14,1 километров, указывающий на северо-восток. Лханаппа, великий охотник на мамонтов, тоже знает, как складывать векторы.

Сложение двух векторов имеет простую геометрическую интерпретацию. Сначала нарисуйте один вектор. Затем нарисуйте второй, начав его со стрелки, которая обозначает окончание первого вектора. В завершение нарисуйте вектор из начальной точки первого вектора к концу второго. Этот третий вектор представляет собой сумму первых двух (рис. 11.2). Следует отметить, что длина результирующего вектора может быть меньше, чем простая сумма длин составляющих его векторов.

Векторы — это естественный выбор для перегрузки операций. Во-первых, вектор нельзя представить одиночным числом, поэтому имеет смысл создать для этого класс. Во-вторых, векторы поддерживают аналоги обычных арифметических операций, таких как сложение и вычитание. Эта параллель предполагает перегрузку соответствующих операций так, чтобы их можно было использовать с векторами.

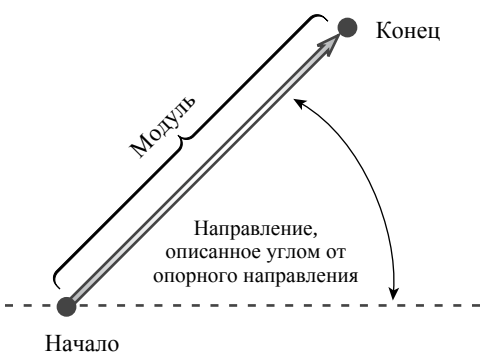


Рис. 11.1. *Описание смещения с помощью вектора*

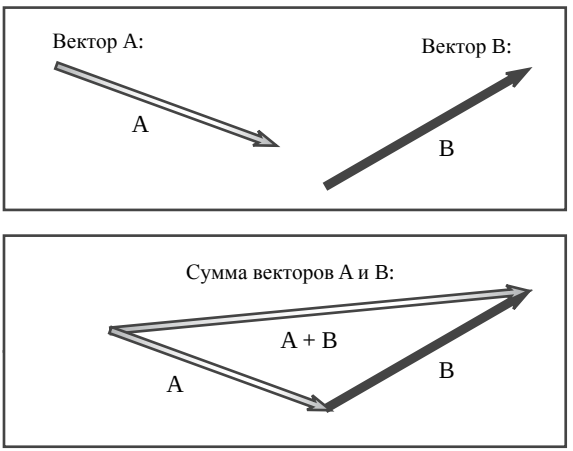


Рис. 11.2. *Сложение двух векторов*

Не особенно усложняя, в настоящем разделе мы реализуем двумерный вектор, такой как экранное смещение, а не трехмерный вектор вроде того, с помощью которого может быть представлено движение вертолета или гимнаста в воздухе. Для описания двумерного вектора понадобятся два числа, однако у вас есть выбор — что именно будут означать эти два числа:

- вектор можно представить модулем (длиной) и направлением (углом);
- вектор можно представить с помощью компонент *x* и *y*.

Компоненты — это горизонтальный вектор (компонент *x*) и вертикальный вектор (компонент *y*), которые в сумме образуют финальный вектор. Например, движение может быть описано как перемещение на 30 единиц вправо и на 40 вверх (рис. 11.3). Это перемещение помещает точку в то же положение, что и перемещение на 50 единиц под углом 53,1 градуса к горизонтали. Таким образом, вектор с модулем 50 и углом 53,1 градуса эквивалентен вектору, имеющему горизонтальный компонент 30 и вертикальный компонент 40. Для вектора смещения существенно знать начальную и конечную точки, а не точный маршрут перемещения из первой точки во вторую. Этот выбор в представлении соответствует рассмотренной в главе 7 программе преобразования между прямоугольными и полярными координатами.

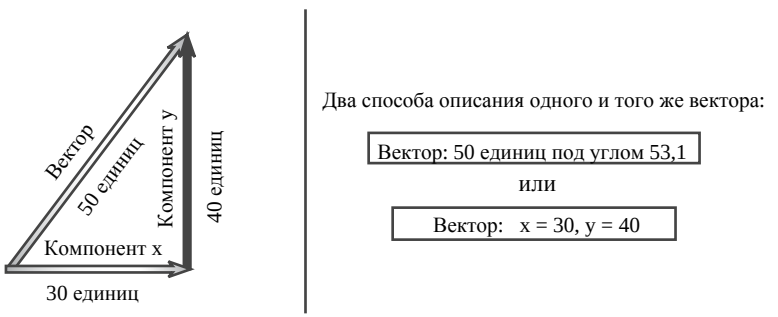


***Рис. 11.3.** Компоненты x и y вектора*

Иногда удобнее одна форма, иногда — другая, поэтому включим в описание класса оба представления. (См. врезку "Множественные представления и классы" далее в этой главе.) Спроектируем класс так, что если вы измените одно из представлений вектора, то второе будет изменяться автоматически. Возможность обеспечить такое интеллектуальное поведение объекта — еще одно преимущество классов C++.

В листинге 11.13 приведено объявление класса. Чтобы освежить ваши знания пространств имен, в этом листинге объявление класса помещено внутрь пространства имен `VECTOR`. Кроме того, в программе используется `enum` с парой констант (`RECT` и `POL`), служащих для идентификации этих двух представлений. (Такой прием уже рассматривался в главе 10, поэтому мы спокойно можем пользоваться им.)

Листинг 11.13. `vect.h`

```
// vect.h -- класс Vector с операцией << и поддержкой режима координат
#ifndef VECTOR_H_
#define VECTOR_H_
#include <iostream>
namespace VECTOR
{
```

```
   class Vector
   {
   public:
      enum Mode {RECT, POL};
      // RECT — для режима прямоугольных координат, POL — для режима полярных координат
   private:
      double x;      // горизонтальное значение
      double y;      // вертикальное значение
      double mag;    // длина вектора
      double ang;    // направление вектора в градусах
      Mode mode;     // RECT или POL

      // Закрытые методы для установки значений
      void set_mag();
      void set_ang();
      void set_x();
      void set_y();
   public:
      Vector();
      Vector(double n1, double n2, Mode form = RECT);
      void reset(double n1, double n2, Mode form = RECT);
      ~Vector();
      double xval() const {return x;}        // сообщает значение x
      double yval() const {return y;}        // сообщает значение y
      double magval() const {return mag;}    // сообщает модуль
      double angval() const {return ang;}    // сообщает угол
      void polar_mode();                     // устанавливает режим в POL
      void rect_mode();                      // устанавливает режим в RECT

      // Перегрузка операций
      Vector operator+(const Vector & b) const;
      Vector operator-(const Vector & b) const;
      Vector operator-() const;
      Vector operator*(double n) const;

      // Друзья
      friend Vector operator*(double n, const Vector & a);
      friend std::ostream &
      operator<<(std::ostream & os, const Vector & v);
   };
} // конец пространства имен VECTOR
#endif
```

В листинге 11.13 обратите внимание, что четыре функции, возвращающие значения компонентов, определены в объявлении класса. Это автоматически делает их встроенными. Ни одна из них не должна изменять данные объекта, поэтому они объявлены с модификатором `const`. Вы можете вспомнить из главы 10, что это синтаксис для объявления функций, не модифицирующих объект, к которому они имеют непосредственный доступ.

В листинге 11.14 показаны все методы и дружественные функции, объявленные в листинге 11.13. Код в листинге использует открытую природу пространств имен для добавления объявлений методов к пространству имен `VECTOR`. Следует отметить, что конструкторы и функция `reset()` устанавливают значения как для прямоугольного, так и для полярного представления вектора. Таким образом, оба набора значений становятся доступными немедленно, без дополнительных вычислений. Кроме того, как упоминалось в главах 4 и 7, встроенные математические функции C++ работают с углами в радианах, поэтому функции, преобразующие значения в градусы и обрат-

но, встроены в методы. Реализация класса Vector скрывает от пользователя такие вещи, как преобразования из полярных координат в прямоугольные и преобразования радианов в градусы. Все, что требуется знать пользователю — это то, что класс использует углы в градусах, и то, что он представляет вектор в двух эквивалентных представлениях.

Листинг 11.14. vect.cpp

```
// vect.cpp -- методы класса Vector
#include <cmath>
#include "vect.h"   // включает <iostream>
using std::sqrt;
using std::sin;
using std::cos;
using std::atan;
using std::atan2;
using std::cout;
namespace VECTOR
{
   // Вычисляет количество градусов в одном радиане
   const double Rad_to_deg = 45.0 / atan(1.0);
   // должно быть приблизительно равно 57.2957795130823

   // Закрытые методы
   // Вычисляет модуль из x и y
   void Vector::set_mag()
   {
      mag = sqrt(x * x + y * y);
   }
   void Vector::set_ang()
   {
      if (x == 0.0 && y == 0.0)
         ang = 0.0;
      else
         ang = atan2(y, x);
   }

   // Устанавливает x по полярным координатам
   void Vector::set_x()
   {
      x = mag * cos(ang);
   }

   // Устанавливает y по полярным координатам
   void Vector::set_y()
   {
      y = mag * sin(ang);
   }

   // Открытые методы
   Vector::Vector() // конструктор по умолчанию
   {
      x = y = mag = ang = 0.0;
      mode = RECT;
   }

   // Конструирует вектор по прямоугольным координатам, если form равно RECT
   // (по умолчанию), или по полярным координатам, если form равно POL
   Vector::Vector(double n1, double n2, Mode form)
   {
      mode = form;
```

```
    if (form == RECT)
    {
        x = n1;
        y = n2;
        set_mag();
        set_ang();
    }
    else if (form == POL)
    {
        mag = n1;
        ang = n2 / Rad_to_deg;
        set_x();
        set_y();
    }
    else
    {
        // Некорректный третий аргумент Vector(); вектор устанавливается в 0
        cout << "Incorrect 3rd argument to Vector() -- ";
        cout << "vector set to 0\n";
        x = y = mag = ang = 0.0;
        mode = RECT;
    }
}
// Устанавливает вектор по прямоугольным координатам, если form равно RECT
// (по умолчанию), или по полярным координатам, если если form равно POL
void Vector:: reset(double n1, double n2, Mode form)
{
    mode = form;
    if (form == RECT)
    {
        x = n1;
        y = n2;
        set_mag();
        set_ang();
    }
    else if (form == POL)
    {
        mag = n1;
        ang = n2 / Rad_to_deg;
        set_x();
        set_y();
    }
    else
    {
        cout << "Incorrect 3rd argument to Vector() -- ";
        cout << "vector set to 0\n";
        x = y = mag = ang = 0.0;
        mode = RECT;
    }
}
Vector::~Vector()          // деструктор
{
}
void Vector::polar_mode()  // устанавливает режим полярных координат
{
    mode = POL;
}
```

```
void Vector::rect_mode()  // устанавливает режим прямоугольных координат
{
   mode = RECT;
}
// Перегрузка операций
// Сложение двух векторов
Vector Vector::operator+(const Vector & b) const
{
   return Vector(x + b.x, y + b.y);
}
// Вычитание вектора b из a
Vector Vector::operator-(const Vector & b) const
{
   return Vector(x - b.x, y - b.y);
}
// Смена знака вектора на противоположный
Vector Vector::operator-() const
{
   return Vector(-x, -y);
}
// Умножение вектора на n
Vector Vector::operator*(double n) const
{
   return Vector(n * x, n * y);
}
// Дружественные методы
// Умножение n на вектор a
Vector operator*(double n, const Vector & a)
{
   return a * n;
}
// Отображает прямоугольные координаты, если mode равно RECT,
// или отображает полярные координаты, если mode равно POL
std::ostream & operator<<(std::ostream & os, const Vector & v)
{
   if (v.mode == Vector::RECT)
      os << "(x,y) = (" << v.x << ", " << v.y << ")";
   else if (v.mode == Vector::POL)
   {
      os << "(m,a) = (" << v.mag << ", "
         << v.ang * Rad_to_deg << ")";
   }
   else
      os << "Vector object mode is invalid"; // недопустимый режим объекта Vector
   return os;
}
} // конец пространства имен VECTOR
```

Спроектировать класс `Vector` можно по-разному. Например, объект может хранить прямоугольные координаты и не хранить полярные. В этом случае вычисление полярных координат может быть помещено в методы `magval()` и `angval()`. Для приложений, в которых преобразование выполняется нечасто, такое решение может оказаться более эффективным. Кроме того, метод `reset()` в этом случае не нужен.

Предположим, что shove — это объект типа Vector, и есть такой код:

```
shove.reset(100, 300);
```

Тот же самый результат можно получить с помощью следующего конструктора:

```
shove = Vector(100, 300);
```

Однако метод reset() изменяет содержимое shove напрямую, в то время как применение конструктора добавляет несколько дополнительных шагов по созданию временного объекта и присваиванию его shove.

Такое проектное решение следует традиции ООП, которая заключается в том, чтобы иметь интерфейс класса, сконцентрированный на сущностях (абстрактную модель), при этом скрывая детали. Таким образом, при использовании класса Vector можно думать об основных свойствах вектора, таких как способность представлять смещение и возможность складывать два вектора. Когда вектор выражается в компонентной нотации либо в нотации модуля и направления, вторая нотация синхронизируется автоматически, и можно в любой момент устанавливать значение вектора и опрашивать его в любом из двух форматов.

Далее мы рассмотрим некоторые из свойств класса Vector более подробно.

Использование члена, хранящего состояние

Класс Vector сохраняет и прямоугольные, и полярные координаты вектора. Его член по имени mode, которая управляет тем, какая форма конструктора, метода reset() и перегруженной операции << будет использоваться. Значение перечисления RECT представляет режим прямоугольных координат (режим по умолчанию), а POL — полярных. Такая переменная-член называется *членом состояния*, поскольку описывает состояние объекта. Чтобы увидеть, что это означает, давайте посмотрим на код конструктора:

```
Vector::Vector(double n1, double n2, Mode form)
{
   mode = form;
   if (form == RECT)
   {
      x = n1;
      y = n2;
      set_mag();
      set_ang();
   }
   else if (form == POL)
   {
      mag = n1;
      ang = n2 / Rad_to_deg;
      set_x();
      set_y();
   }
   else
   {
      // Некорректный третий аргумент Vector(); вектор устанавливается в 0
      cout << "Incorrect 3rd argument to Vector() -- ";
      cout << "vector set to 0\n";
      x = y = mag = ang = 0.0;
      mode = RECT;
   }
}
```

Если третий аргумент имеет значение RECT или же он опущен (в этом случае прототип присваивает значение RECT по умолчанию), то входные параметры интерпретируются как прямоугольные координаты, в то время как значение POL заставляет интерпретировать их как полярные координаты:

```
Vector folly(3.0, 4.0);                          // установить x = 3, y = 4
Vector foolery(20.0, 30.0, VECTOR::Vector::POL); // установить mag = 20, ang = 30
```

Идентификатор POL имеет область видимости класса, поэтому внутри определений этого класса можно просто указывать его неуточненное имя. Полностью уточненное имя выглядит как VECTOR::Vector::POL, поскольку идентификатор POL определен в классе Vector, а сам Vector определен в пространстве имен VECTOR. Обратите внимание, что конструктор использует закрытые методы set_mag() и set_ang() для установки значений модуля и угла, если переданы значения x и y, и закрытые методы set_x() и set_y() для установки значений x и y, если переданы значения модуля и угла. Также следует отметить, что конструктор выдает предупреждающее сообщение и устанавливает состояние в RECT, если указано что-либо отличное от RECT и POL.

Сейчас может показаться довольно затруднительным передать конструктору что-то, отличающееся от RECT или POL, поскольку третий аргумент имеет тип VECTOR::Vector::Mode. Вызов, подобный приведенному ниже, не скомпилируется, т.к. целочисленное значение вроде 2 не может быть неявно преобразовано к типу enum:

```
Vector rector(20.0, 30.0, 2); // несоответствие типа — 2 не относится к типу enum
```

Тем не менее, находчивый и любопытный пользователь может поступить следующим образом и посмотреть, что произойдет:

```
Vector rector(20.0, 30.0, VECTOR::Vector::Mode (2));  // приведение типа
```

В этом случае компиляция пройдет нормально.

Далее функция operator<<() использует член mode для определения того, как отобразить значения:

```
// Отображает прямоугольные координаты, если mode равно RECT,
// или отображает полярные координаты, если mode равно POL
std::ostream & operator<<(std::ostream & os, const Vector & v)
{
    if (v.mode == Vector::RECT)
      os << "(x,y) = (" << v.x << ", " << v.y << ")";
    else if (v.mode == Vector::POL)
    {
      os << "(m,a) = (" << v.mag << ", "
         << v.ang * Rad_to_deg << ")";
    }
    else
      os << "Vector object mode is invalid"; // недопустимый режим объекта Vector
    return os;
}
```

Поскольку operator<<() — это дружественная функция, не относящаяся к области видимости класса, необходимо использовать Vector::RECT, а не просто RECT. Но функция находится в пространстве имен VECTOR, поэтому применять полностью уточненное имя VECTOR::Vector::RECT не нужно.

Различные методы, которые могут устанавливать режим, заботятся о том, чтобы в качестве допустимых значений принимались только RECT и POL, поэтому последняя конструкция else в этой функции никогда не должна быть достигнута. Однако, не-

смотря на это, проверять все же стоит: такая проверка во многих случаях поможет перехватить некоторые неочевидные ошибки.

Множественные представления и классы

Величины, которые имеют различное, однако эквивалентное представление, встречаются часто. Например, расход топлива можно измерять в милях на галлон, как это делается в США, или же в литрах на 100 километров, как принято в Европе. Число может быть представлено в строковой или числовой форме и т.д. Классы хорошо приспосабливаются для представления разных аспектов существования в одном объекте. Во-первых, в одном объекте можно сохранять множество представлений. Во-вторых, функции класса могут быть написаны так, что присваивание значений для одного представления автоматически установит значения для других представлений. Например, метод `set_by_polar()` в классе `Vector` устанавливает члены `mag` и `ang` в значения аргументов функции, но также устанавливает члены `x` и `y`. Или же можно хранить данные в единственном представлении и с помощью методов сделать доступными другие представления. За счет внутренней поддержки преобразований класс поможет думать о величине в терминах ее природы, а не в терминах представления.

Перегрузка арифметических операций для класса `Vector`

Сложение двух векторов очень просто, если используются координаты `x` и `y`. Для получения результирующих значений `x` и `y` нужно просто просуммировать между собой попарно два компонента `x` и два компонента `y`. Исходя из такого описания, можно предположить, что код должен выглядеть следующим образом:

```
Vector Vector::operator+(const Vector & b) const
{
   Vector sum;
   sum.x = x + b.x;
   sum.y = y + b.y;
   return sum;          // незавершенная версия
}
```

И все было бы в порядке, если бы в объекте хранились только компоненты `x` и `y`. К сожалению, эта версия кода не поддерживает установку значений полярных координат. Решить эту проблему можно добавлением нескольких строк:

```
Vector Vector::operator+(const Vector & b) const
{
   Vector sum;
   sum.x = x + b.x;
   sum.y = y + b.y;
   sum.set_ang(sum.x, sum.y);
   sum.set_mag(sum.x, sum.y);
   return sum;          // в этой версии присутствует ненужное дублирование
}
```

Но проще и удобнее позволить выполнять эту работу конструктору:

```
Vector Vector::operator+(const Vector & b) const
{
  return Vector(x + b.x, y + b.y); // возвращает сконструированный объект Vector
}
```

Здесь в коде используется конструктор `Vector` для установки значений компонентов `x` и `y`. Конструктор затем создает новый безымянный объект, и функция возвращает этот объект. Подобным образом гарантируется, что новый объект будет создан в соответствии со стандартными правилами, заложенными в конструктор.

Совет

Если метод должен вычислять новый объект класса, вы должны посмотреть, нельзя ли для выполнения этой работы воспользоваться конструктором класса. Это не только избавит от забот, но также гарантирует, что новый объект будет сконструирован должным образом.

Умножение

В визуальных терминах умножение вектора на число делает его длиннее или короче в заданное число раз. Поэтому умножение вектора на 3 создает вектор втрое большей длины, чем исходный, но с тем же самым направлением. Перенести это на представление вектора с помощью класса `Vector` довольно просто. В терминах полярных координат осуществляется умножение длины без изменения угла. В терминах прямоугольных координат умножение вектора на число означает отдельное умножение на это число каждого из компонентов `x` и `y`. То есть, если вектор имеет компоненты `x` и `y`, равные 5 и 12, умножение их на 3 дает им значения 15 и 36. Именно это и делает перегруженная операция умножения:

```
Vector Vector::operator*(double n) const
{
   return Vector(n * x, n * y);
}
```

Как и в случае с перегруженной операцией сложения, этот код позволяет конструктору создать корректный объект `Vector` на основе новых компонентов `x` и `y`. Приведенный код поддерживает умножение значения `Vector` на значение `double`. Точно так же, как в примере с `Time`, для умножения `Vector` на `double` можно использовать встроенную дружественную функцию:

```
Vector operator*(double n, const Vector & a)  // дружественная функция
{
   return a * n;      // преобразует умножение double на Vector
                      // в умножение Vector на double
}
```

Дополнительное усовершенствование: перегрузка перегруженной операции

В языке C++ операция — уже имеет две трактовки. Во-первых, когда используется с двумя операндами, она является операцией вычитания. Операция вычитания называется *бинарной операцией*, поскольку она работает с двумя операндами. Во-вторых, когда операция — применяется с одним операндом, как в `-x`, она является операцией смены знака. Такая форма называется *унарной операцией*, что означает наличие только одного операнда. Как вычитание, так и смена знака имеют смысл и для вектора, поэтому в классе `Vector` присутствуют они обе.

Чтобы вычесть вектор B из вектора A, необходимо просто обеспечить вычитание компонентов, поэтому определение перегруженного вычитания очень похоже на сложение:

```
Vector operator-(const Vector & b) const;              // прототип
Vector Vector::operator-(const Vector & b) const       // определение
{
   return Vector(x - b.x, y - b.y); // возвращает сконструированный объект Vector
}
```

Здесь важно правильно указать порядок. Рассмотрим следующий оператор:

```
diff = v1 - v2;
```

Он преобразуется в такой вызов функции-члена:

```
diff = v1.operator-(v2);
```

Это означает, что вектор, который передается как явный аргумент, вычитается из вектора, переданного неявным аргументом, поэтому необходимо использовать `x - b.x`, а не `b.x - x`.

Далее рассмотрим унарную операцию `-`, которая принимает только один операнд. Применение этой операции к обычному числу, как в `-x`, меняет знак значения. Таким образом, применение этой операции к вектору должно менять знак каждого компонента. Точнее говоря, функция должна возвращать новый вектор, противоположный исходному. (В терминах полярных координат отрицание оставляет модуль вектора без изменений, но меняет направление на противоположное.) Ниже представлен прототип и определение перегруженного отрицания:

```
Vector operator-() const;
Vector Vector::operator-() const
{
   return Vector (-x, -y);
}
```

Обратите внимание, что теперь есть два разных определения `operator-()`. И это нормально, поскольку эти два определения имеют разные сигнатуры. Определить унарную и бинарную версии операции — можно потому, что в C++ изначально представлены две версии. Операция, которая имеет только бинарную форму, такая как деление (`/`), может быть перегружена только как бинарная.

На заметку!

Поскольку перегрузка операций реализуется с помощью функций, одну и ту же операцию можно перегружать много раз, при условии, что каждая функция операции имеет отличную от других сигнатуру, и каждая функция операции поддерживает то же количество операндов, что и соответствующая встроенная операция C++.

Комментарии к реализации

Реализация, описанная в предыдущих разделах, сохраняет и прямоугольные, и полярные координаты вектора в объекте `Vector`. Однако открытый интерфейс не зависит от этого факта. Интерфейс требует только то, что оба представления могут быть отображены, а индивидуальные значения возвращены. Внутренняя реализация может существенно отличаться. Как упоминалось ранее, объект может сохранять только компоненты `x` и `y`. Затем, скажем, метод `magval()`, который возвращает значение модуля вектора, может вычислить модуль на основе значений `x` и `y` вместо того, чтобы обращаться к значению, которое хранится в объекте как отдельный компонент. Такой подход изменяет реализацию, но оставляет интерфейс неизменным. Подобное отделение интерфейса от реализации является одной из целей ООП. Оно позволяет тонко настроить реализацию без изменения кода в программах, использующих класс.

Оба варианта реализации обладают своими преимуществами и недостатками. Сохранение данных означает, что объект занимает больше места в памяти и что код должен заботиться о синхронном обновлении и прямоугольного, и полярного представлений всякий раз, когда объект `Vector` изменяется. Но поиск данных выполняется быстрее. Если приложение часто нуждается в доступе к обоим представлениям вектора, то предпочтительна реализация, показанная в примере. Если же полярное представление требуется только изредка, лучше остановиться на втором представлении. Вы можете выбрать одну реализацию для одной программы и другую — для другой, используя один и тот же интерфейс для обоих.

Использование класса `Vector` при решении задачи случайного блуждания

В листинге 11.15 приведена короткая программа, которая использует усовершенствованный класс `Vector`. Она моделирует известную задачу случайного блуждания. Идея заключается в том, что вы помещаете кого-то в исходную точку. Он начинает двигаться, но направление с каждым шагом случайным образом изменяется по отношению к направлению на предыдущем шаге. Вот как выглядит одна из формулировок этой задачи: сколько шагов нужно сделать этому персонажу, чтобы удалиться, скажем, на 50 футов от исходной точки? В терминах векторов это означает сложение множества случайно ориентированных векторов до тех пор, пока сумма не превысит 50 футов.

Код в листинге 11.15 позволяет выбрать заданное расстояние, которое нужно преодолеть, и длину шага. Он поддерживает промежуточную сумму, которая представляет позицию после каждого шага (в виде вектора), и сообщает количество шагов, необходимых для преодоления заданного расстояния в соответствии с текущим положением (в обоих форматах). Как вы увидите, перемещение персонажа достаточно неэффективно. Путешествие из 1000 шагов, по 2 фута каждый, может увести его всего на 50 футов от начальной точки. Для измерения степени неэффективности общая преодоленная дистанция (в данном случае 50 футов) делится на количество шагов. Все случайные изменения направления делают среднюю длину перемещения значительно меньше, чем один шаг. Для случайного выбора направления в программе используются стандартные библиотечные функции `rand()`, `srand()` и `time()`, которые описаны в следующем разделе "Замечания по программе". Обеспечьте совместную компиляцию кода из листингов 11.15 и 11.14.

Листинг 11.15. `randwalk.cpp`

```
// randwalk.cpp — использование класса Vector
// Компилировать вместе с файлом vect.cpp
#include <iostream>
#include <cstdlib>       // прототипы rand(), srand()
#include <ctime>         // прототип time()
#include "vect.h"
int main()
{
   using namespace std;
   using VECTOR::Vector;
   srand(time(0));       // начальное значение для генератора случайных чисел
   double direction;
   Vector step;
   Vector result(0.0, 0.0);
   unsigned long steps = 0;
   double target;
   double dstep;
   cout << "Enter target distance (q to quit): ";
         // Ввод заданного расстояния (q для завершения)
   while (cin >> target)
   {
      cout << "Enter step length: "; // ввод длины шага
      if (!(cin >> dstep))
         break;
      while (result.magval() < target)
      {
         direction = rand() % 360;
         step.reset(dstep, direction, Vector::POL);
```

```
            result = result + step;
            steps++;
        }
        cout << "After " << steps << " steps, the subject "
             "has the following location:\n";
        cout << result << endl; // вывод позиции после steps шагов
        result.polar_mode();
        cout << " or\n" << result << endl;
        cout << "Average outward distance per step = "
             << result.magval()/steps << endl; // вывод среднего расстояния на один шаг
        steps = 0;
        result.reset(0.0, 0.0);
        cout << "Enter target distance (q to quit): ";
             // Ввод заданного расстояния (q для завершения)
    }
    cout << "Bye!\n";
    cin.clear();
    while (cin.get() != '\n')
        continue;
    return 0;
}
```

Поскольку в программе присутствует объявление `using`, помещающее `Vector` в область видимости, можно использовать `Vector::POL` вместо `VECTOR::Vector::POL`.

Ниже показан пример выполнения программы из листингов 11.13, 11.14 и 11.15:

```
Enter target distance (q to quit): 50
Enter step length: 2
After 253 steps, the subject has the following location:
(x,y) = (46.1512, 20.4902)
or
(m,a) = (50.495, 23.9402)
Average outward distance per step = 0.199587
Enter target distance (q to quit): 50
Enter step length: 2
After 951 steps, the subject has the following location:
(x,y) = (-21.9577, 45.3019)
or
(m,a) = (50.3429, 115.8593)
Average outward distance per step = 0.0529362
Enter target distance (q to quit): 50
Enter step length: 1
After 1716 steps, the subject has the following location:
(x,y) = (40.0164, 31.1244)
or
(m,a) = (50.6956, 37.8755)
Average outward distance per step = 0.0295429
Enter target distance (q to quit): q
Bye!
```

Случайная природа этого процесса порождает различные вариации от попытки к попытке, даже если начальные условия одинаковы. Однако в среднем уменьшение в два раза размера шага учетверяет количество шагов, необходимых для преодоления дистанции. Согласно теории вероятностей, среднее количество шагов (N) длиной s, которое понадобится для преодоления суммарного расстояния D, вычисляется по следующей формуле:

$$N = (D/s)^2$$

Но это среднее значение, которое будет существенно варьироваться от попытки к попытке. Например, 1000 попыток преодоления 50 футов при 2-футовом шаге в среднем дают 636 шагов (что близко к теоретическому значению 625) на прохождение этого расстояния, но оно колеблется в диапазоне от 91 до 3951. Соответственно, 1000 попыток преодоления 50 футов при 1-футовом шаге дают в среднем 2557 шагов (близко к теоретическому значению 2500) с диапазоном от 345 до 10882. Итак, если вам придется двигаться случайным образом, знайте, что лучше идти большими шагами. Вы никак не сможете повлиять на выбор направления, но, по крайней мере, уйдете дальше.

Замечания по программе

Первым делом, в листинге 11.15 обратите внимание, насколько легко используется пространство имен VECTOR. Следующее объявление помещает имя класса `Vector` в область видимости:

```
using VECTOR::Vector;
```

Поскольку все методы класса `Vector` имеют область видимости класса, импортирование имени класса также делает все методы класса `Vector` доступными, без необходимости применения дополнительных объявлений `using`.

Далее поговорим о случайных числах. Стандартная библиотека ANSI C, которая также поставляется вместе с C++, включает в себя функцию `rand()`, которая возвращает случайное целое число в диапазоне от нуля до определенного значения, зависящего от реализации. Программа моделирования случайного блуждания использует операцию взятия модуля для выбора угла направления шага в диапазоне от 0 до 359. Функция `rand()` работает, применяя свой алгоритм к начальному значению для получения очередного случайного числа. Это число используется как начальное значение при следующем вызове функции и т.д. На самом деле получается ряд *псевдослучайных* чисел, поскольку 10 последовательных вызовов обычно генерируют один и тот же набор 10 случайных чисел. (Конкретные значения зависят от реализации.) Однако функция `srand()` позволяет изменить начальное значение и получить другую последовательность случайных чисел. Для инициализации генератора в программе используется значение, возвращенное `time(0)`. Вызов `time(0)` возвращает текущее время, часто реализованное в виде количества секунд, прошедших с определенной специфической даты. (В общем случае `time()` принимает адрес переменной типа `time_t`, помещает в нее значение времени и также возвращает ее. Использование `0` в качестве аргумента-адреса исключает потребность в переменной типа `time_t`.) Таким образом, следующий оператор устанавливает разное начальное значение при каждом запуске программы, обеспечивая еще более случайную последовательность чисел:

```
srand(time(0));
```

Заголовочный файл `cstdlib` (бывший `stdlib.h`) содержит прототипы функций `srand()` и `rand()`, а `ctime` (бывший `time.h`) — прототип `time()`.(C++11 предоставляет более развитую поддержку генерации случайных чисел в виде функций из заголовочного файла `random`.)

Программа использует вектор `result` для того, чтобы отслеживать случайное движение. При каждом проходе вложенного цикла для вектора `step` устанавливается новое направление, которое добавляется к текущему значению вектора `result`. Когда величина вектора `result` превысит заданную дистанцию, цикл завершается.

За счет установки режима вектора программа отображает конечную позицию в прямоугольных и полярных координатах.

Кстати, следующий оператор переключает result в режим RECT, независимо от начальных режимов result и step:

```
result = result + step;
```

Давайте посмотрим, почему так. Сначала функция операции сложения создает и возвращает новый вектор, представляющий сумму двух аргументов. Функция создает вектор, используя конструктор по умолчанию, который порождает вектор в режиме RECT. Поэтому вектор, присваиваемый result, находится в режиме RECT. По умолчанию присваивание каждой переменной-члена выполняется индивидуально, поэтому result.mode получает значение RECT. Если вы предпочитаете какое-то другое поведение, например, чтобы result сохранял свой предыдущий режим, можете перегрузить операцию присваивания по умолчанию, определив для класса функцию операции присваивания. Примеры такого подхода приведены в главе 12.

Сохранять позицию в файле очень просто. Первым делом необходимо включить <fstream>, объявить объект ofstream и ассоциировать его с файлом:

```
#include <fstream>
...
ofstream fout;
fout.open("thewalk.txt");
```

Затем в цикл, вычисляющий результат, понадобится вставить примерно такой оператор:

```
fout << result << endl;
```

Это вызывает дружественную функцию operator<<(fout, result), подставляя в качестве первого аргумента ссылку на fout и, таким образом, направляя вывод в файл. Объект fout можно также использовать для вывода в этот файл другой информации, такой как итоговые сведения, отображаемые cout.

Автоматические преобразования и приведения типов в классах

Следующей темой при обсуждении классов будет преобразование типов. Мы посмотрим, как C++ поддерживает преобразования в определяемые пользователем типы и обратно. Для начала мы рассмотрим поддержку в C++ преобразований для встроенных типов. Когда есть оператор, который присваивает значение одного стандартного типа переменной другого стандартного типа, C++ автоматически преобразует присваиваемое значение в тип принимающей переменной, при условии, что эти два типа совместимы. Например, все приведенные ниже операторы генерируют преобразования числовых типов:

```
long count = 8;       // значение 8 типа int преобразуется в тип long
double time = 11;     // значение 11 типа int преобразуется в тип double
int side = 3.33;      // значение 3.33 типа double преобразуется в тип int (3)
```

Эти присваивания работают, т.к. C++ распознает, что все эти разнообразные типы представляют одну и ту же базовую сущность — число, а также потому, что C++ включает встроенные правила для выполнения преобразований между ними. Однако вспомните главу 3, где говорилось о том, что при таком преобразовании может быть потеряна точность. Например, присваивание значения 3.33 переменной side типа int дает в результате значение 3, с потерей дробной части 0.33.

В C++ несовместимые между собой типы автоматически не преобразуются. Например, следующий оператор завершится сбоем, поскольку слева от знака равенства находится указатель, а справа — число:

```
int * p = 10;              // конфликт типов
```

И даже хотя компьютер может внутренне представлять адрес с помощью целого числа, концептуально адреса и целые числа являются совершенно разными. Например, указатель нельзя возводить в квадрат. Однако когда автоматическое преобразование не срабатывает, можно использовать приведение типа:

```
int * p = (int *) 10;      // нормально, p и (int *) 10 — оба указатели
```

Этот код устанавливает указатель в адрес 10, выполнив приведение 10 к типу указателя на int (т.е. к типу int *). Имеет ли подобное присваивание смысл — это другой вопрос.

Если определяемый класс в достаточной мере связан с базовым типом или другим классом, преобразование одного в другой имеет смысл. В этом случае можно указать C++, как выполнять преобразование автоматически либо, возможно, через приведение типа. Чтобы посмотреть, как это работает, перепишем программу преобразования стоунов в фунты из главы 3 с использованием класса. Сначала необходимо спроектировать подходящий тип. По существу нужно представить одно и то же понятие (вес) двумя способами (в фунтах и стоунах). Класс предлагает отличный способ включить два представления одной концепции в одну сущность. Таким образом, имеет смысл поместить оба представления веса в один класс и затем предусмотреть методы для выражения веса в различной форме. В листинге 11.16 показан заголовок этого класса.

Листинг 11.16. stonewt.h

```
// stonewt.h -- определение класса Stonewt
#ifndef STONEWT_H_
#define STONEWT_H_
class Stonewt
{
private:
    enum {Lbs_per_stn = 14};          // фунтов на стоун
    int stone;                        // полных стоунов
    double pds_left;                  // дробное число фунтов
    double pounds;                    // общий вес в фунтах
public:
    Stonewt(double lbs);              // конструктор для значения в фунтах
    Stonewt(int stn, double lbs);     // конструктор для значения в стоунах и фунтах
    Stonewt();                        // конструктор по умолчанию
    ~Stonewt();
    void show_lbs() const;            // отображение веса в формате фунтов
    void show_stn() const;            // отображение веса в формате стоунов
};
#endif
```

Как упоминалось в главе 10, enum предоставляет удобный способ определения специфичных для класса констант, при условии, что они будут целочисленными. Можно также воспользоваться следующей альтернативой:

```
static const int Lbs_per_stn = 14;
```

Обратите внимание, что класс Stonewt имеет три конструктора. Они позволяют инициализировать объект Stonewt числом с плавающей точкой для фунтов или ком-

бинацией стоунов и фунтов. Либо же можно создать объект Stonewt без его инициализации:

```
Stonewt blossem(132.5);       // вес составляет 132.5 фунта
Stonewt buttercup(10, 2);     // вес составляет 10 стоунов, 2 фунта
Stonewt bubbles;              // вес равен значению по умолчанию
```

Этот класс в действительности не нуждается в объявлении деструктора, поскольку автоматически определяемого деструктора по умолчанию в данном случае вполне достаточно. С другой стороны, предоставление явного объявления упростит определение деструктора в будущем, когда это понадобится.

Кроме того, класс Stonewt предоставляет две функции отображения. Одна отображает вес в фунтах, другая — в стоунах и фунтах. В листинге 11.17 показана реализация методов класса. Обратите внимание, что каждый конструктор присваивает значения всем трем закрытым членам. Таким образом, при создании объекта Stonewt автоматически устанавливаются оба представления веса.

Листинг 11.17. stonewt.cpp

```
// stonewt.cpp -- методы класса Stonewt
#include <iostream>
using std::cout;
#include "stonewt.h"

// Конструирует объект Stonewt из значения типа double
Stonewt::Stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;      // целочисленное деление
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
}

// Конструирует объект Stonewt из стоунов и значения типа double
Stonewt::Stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds = stn * Lbs_per_stn +lbs;
}

Stonewt::Stonewt()                        // конструктор по умолчанию, wt = 0
{
    stone = pounds = pds_left = 0;
}

Stonewt::~Stonewt()                       // деструктор
{
}

// Отображение веса в стоунах
void Stonewt::show_stn() const
{
    cout << stone << " stone, " << pds_left << " pounds\n";
}

// Отображение веса в фунтах
void Stonewt::show_lbs() const
{
    cout << pounds << " pounds\n";
}
```

Поскольку объект Stonewt представляет единственный вес, имеет смысл предусмотреть способы для преобразования целочисленного или значения с плавающей точкой в объект Stonewt. И это уже сделано. В C++ любой конструктор, который принимает единственный аргумент, действует как инструмент копирования для преобразования значения типа аргумента в тип класса. Следующий конструктор служит инструкциями для преобразования значения типа double в значение типа Stonewt:

```
Stonewt(double lbs);      // шаблон преобразования double в Stonewt
```

То есть можно записать такой код:

```
Stonewt myCat;            // создание объекта Stonewt
myCat = 19.6;             // использование Stonewt(double)
                          // для преобразования 19.6 в Stonewt
```

Программа использует конструктор Stonewt(double) для построения временного объекта Stonewt с указанием 19.6 в качестве инициализирующего значения. Затем операция почленного присваивания копирует содержимое временного объекта в myCat. Этот процесс известен как *неявное преобразование*, поскольку происходит автоматически, без необходимости в явном приведении типов.

В качестве функции преобразования может использоваться только конструктор с одним аргументом.

Следующий конструктор принимает два аргумента, поэтому применяться для преобразования типов не может:

```
Stonewt(int stn, double lbs);      // не является функцией преобразования
```

Однако если предусмотреть в нем значение по умолчанию для второго параметра, он сможет действовать как руководство для преобразования int:

```
Stonewt(int stn, double lbs = 0);  // преобразование int в Stonewt
```

Возможность применения конструктора, работающего как автоматическая функция преобразования типов, кажется удобным средством. Но программисты, накопившие определенный опыт работы с C++, обнаруживают, что автоматический аспект не всегда желателен, поскольку иногда ведет к неожиданным преобразованиям. Поэтому в C++ добавлено новое ключевое слово explicit для отключения этого автоматического поведения. Значит, конструктор можно объявить следующим образом:

```
explicit Stonewt(double lbs);      // неявное преобразование не разрешено
```

Это отключает неявное преобразование, подобное тому, что приведено в предыдущем примере, но по-прежнему позволяет использовать явное преобразование, т.е. с явными приведениями типов:

```
Stonewt myCat;            // создание объекта Stonewt
myCat = 19.6;             // не допускается, если Stonewt(double)
                          // объявлен как explicit
mycat = Stonewt(19.6);    // так можно, явное преобразование
mycat = (Stonewt) 19.6;   // так можно, это старая форма приведения типов
```

На заметку!

Конструктор C++, который принимает один аргумент, определяет преобразование типа аргумента в тип класса. Если конструктор снабжен ключевым словом explicit, он может использоваться только с явной формой преобразования, в противном случае допускается неявное преобразование.

Когда компилятор использует функцию Stonewt(double)? Если ключевое слово explicit присутствует в объявлении, Stonewt(double) применяется только с явным приведением типов, а в противном случае его можно использовать для следующих неявных преобразований:

- когда объект Stonewt инициализируется значением типа double;
- когда объекту Stonewt присваивается значение типа double;
- когда функции, ожидающей аргумент типа Stonewt, передается значение типа double;
- когда функция, объявленная как возвращающая значение Stonewt, пытается вернуть значение double;
- когда в любой из описанных выше ситуаций используется встроенный тип, который может быть однозначно преобразован в тип double.

Рассмотрим последний пункт более подробно. Процесс сопоставления аргументов, поддерживаемый прототипированием функций, позволяет конструктору Stonewt(double) выступать в качестве преобразователя для других числовых типов. То есть оба следующих оператора работают, преобразуя сначала int в double, а затем обращаясь к конструктору Stonewt(double):

```
Stonewt Jumbo(7000);  // использует Stonewt(double), преобразуя int в double
Jumbo = 7300;         // использует Stonewt(double), преобразуя int в double
```

Однако это двухшаговое преобразование работает только в том случае, когда выбор однозначен. То есть, если у класса также определен конструктор Stonewt(long), то компилятор отклонит эти выражения, возможно, указав, что int может быть преобразован и в double, и в long, поэтому такой вызов неоднозначен.

Код в листинге 11.18 использует два конструктора для инициализации объектов Stonewt и управления преобразованием типов. Обеспечьте совместную компиляцию кода в листингах 11.17 и 11.18.

Листинг 11.18. stone.cpp

```
// stone.cpp -- определенные пользователем преобразования
// Компилировать вместе с stonewt.cpp
#include <iostream>
using std::cout;
#include "stonewt.h"
void display(const Stonewt & st, int n);
int main()
{
    Stonewt incognito = 275; // использование конструктора для инициализации
    Stonewt wolfe(285.7);    // то же, что и Stonewt wolfe = 285.7;
    Stonewt taft(21, 8);
    cout << "The celebrity weighed ";
    incognito.show_stn();
    cout << "The detective weighed ";
    wolfe.show_stn();
    cout << "The President weighed ";
    taft.show_lbs();
    incognito = 276.8;       // использование конструктора для преобразования
    taft = 325;              // то же, что и taft = Stonewt(325);
    cout << "After dinner, the celebrity weighed ";
    incognito.show_stn();
    cout << "After dinner, the President weighed ";
```

```
    taft.show_lbs();
    display(taft, 2);
    cout << "The wrestler weighed even more.\n";
    display(422, 2);
    cout << "No stone left unearned\n";
    return 0;
}
void display(const Stonewt & st, int n)
{
    for (int i = 0; i < n; i++)
    {
        cout << "Wow! ";
        st.show_stn();
    }
}
```

Ниже показан вывод программы из листинга 11.18:

```
The celebrity weighed 19 stone, 9 pounds
The detective weighed 20 stone, 5.7 pounds
The President weighed 302 pounds
After dinner, the celebrity weighed 19 stone, 10.8 pounds
After dinner, the President weighed 325 pounds
Wow! 23 stone, 3 pounds
Wow! 23 stone, 3 pounds
The wrestler weighed even more.
Wow! 30 stone, 2 pounds
Wow! 30 stone, 2 pounds
No stone left unearned
```

Замечания по программе

Обратите внимание, что когда конструктор принимает единственный аргумент, можно использовать следующую форму инициализации объекта класса:

```
// Синтаксис для инициализации объекта класса
// при использовании конструктора с одним аргументом
Stonewt incognito = 275;
```

Это эквивалентно следующим двум формам, которые были показаны ранее:

```
// Стандартные формы синтаксиса для инициализации объектов
Stonewt incognito(275);
Stonewt incognito = Stonewt(275);
```

Однако последние две формы могут применяться с конструкторами, принимающими несколько аргументов.

Далее отметим следующие два присваивания из листинга 11.18:

```
incognito = 276.8;
taft = 325;
```

Первое из двух присваиваний использует конструктор с аргументом типа `double` для преобразования `276.8` в значение типа `Stonewt`. При этом члену `pounds` объекта `incognito` присваивается значение `276.8`. Поскольку здесь применяется конструктор, такое присваивание также устанавливает значение членов `stone` и `pds_left` класса. Аналогично, второе присваивание преобразует значение типа `int` в тип `double` и затем использует `Stonewt(double)` для установки значений всех трех членов класса.

И, наконец, обратим внимание на следующий вызов функции:

```
display(422, 2);  // преобразует 422 в double, затем в Stonewt
```

Прототип функции `display()` указывает на то, что первый аргумент должен быть объектом типа `Stonewt`. (Аргументу `Stonewt` соответствует формальный параметр `Stonewt` или `Stonewt &`.) Обнаружив аргумент типа `int`, компилятор ищет конструктор `Stonewt(int)` для преобразования аргумента `int` в тип `Stonewt`. Не найдя его, компилятор ищет конструктор с аргументом другого встроенного типа, который можно преобразовать в `int`. Конструктор `Stonewt(double)` подходит. Поэтому компилятор преобразует `int` в `double` и затем использует `Stonewt(double)` для преобразования результата в объект `Stonewt`.

Функции преобразования

Код в листинге 11.18 преобразует число в объект `Stonewt`. Возможен ли обратный процесс? Другими словами, можно ли преобразовать объект `Stonewt` в `double`, как в следующем примере:

```
Stonewt wolfe(285.7);
double host = wolfe;  // ?? возможно ли это ??
```

Ответ: это сделать можно, но не за счет использования конструктора. Конструкторы применяются только для преобразования других типов *в* тип класса. Чтобы выполнить обратный процесс, необходимо предусмотреть специальную форму функции операции C++, которая называется *функцией преобразования*.

Функции преобразования — это определяемый пользователем способ приведения типов, и его можно применять таким же способом, как и обычное приведение типов. Например, если определена функция преобразования `Stonewt` в `double`, то можно выполнять следующие преобразования:

```
Stonewt wolfe(285.7);
double host = double (wolfe);          // синтаксис #1
double thinker = (double) wolfe;       // синтаксис #2
```

Либо можно предоставить компилятору решить, что делать:

```
Stonewt wells(20, 3);
double star = wells;  // неявное применение функции преобразования
```

Компилятор, отметив, что правая часть выражения имеет тип `Stonewt`, а левая — `double`, смотрит, определена ли соответствующая описанию функция преобразования. (Если он не находит ее, то генерирует сообщение об ошибке, говорящее о невозможности присваивания значения типа `Stonewt` переменной типа `double`.)

Так каким же образом создать функцию преобразования? Для преобразования в тип *имяТипа* используется функция такого вида:

```
operator имяТипа();
```

При этом нужно помнить о следующих моментах:

- функция преобразования должна быть методом класса;
- в функции преобразования не должен быть указан возвращаемый тип;
- функция преобразования не должна иметь аргументов.

Например, функция для преобразования в тип `double` должна иметь следующий прототип:

```
operator double();
```

Часть *имяТипа* (в данном случае — double) говорит о том, в какой тип нужно преобразовать, и потому никакого типа возврата не требуется. Тот факт, что функция является методом класса, означает, что она должна вызываться конкретным объектом класса, и сообщает ей, какое значение необходимо преобразовать. Поэтому такая функция не нуждается в аргументах.

Чтобы добавить функции, которые преобразуют объект stone_wt в типы int и double, необходимо дополнить объявление класса следующими прототипами:

```
operator int();
operator double();
```

В листинге 11.19 представлена модифицированная версия объявления класса.

Листинг 11.19. stonewt1.h

```
// stonewt1.h -- усовершенствованное определение класса Stonewt
#ifndef STONEWT1_H_
#define STONEWT1_H_
class Stonewt
{
private:
    enum {Lbs_per_stn = 14};        // фунтов на стоун
    int stone;                      // полных стоунов
    double pds_left;                // дробное число фунтов
    double pounds;                  // общий вес в фунтах
public:
    Stonewt(double lbs);            // конструктор для значения double фунтов
    Stonewt(int stn, double lbs);   // конструктор для значения в стоунах и фунтах
    Stonewt();                      // конструктор по умолчанию
    ~Stonewt();
    void show_lbs() const;          // отображение веса в формате фунтов
    void show_stn() const;          // отображение веса в формате стоунов
    // Функции преобразования
    operator int() const;
    operator double() const;
};
#endif
```

Код в листинге 11.20 представляет собой модифицированную версию кода из листинга 11.17 с включением функций преобразования типа. Обратите внимание, что обе функции возвращают значение нужного типа, даже несмотря на то, что не имеют объявленного типа возврата. Кроме того, определение преобразования в int округляет возвращаемое значение к ближайшему целому, а не усекает его. Например, если pounds равно 114.4, то pounds + 0.5 равно 114.9, а int(114.9) равен 114. Но если pounds равно 114.6, то pounds + 0.5 равно 115.1, а int(115.1) равно 115.

Листинг 11.20. stonewt1.cpp

```
// stonewt1.cpp -- методы класса Stonewt с функциями преобразования
#include <iostream>
using std::cout;
#include "stonewt1.h"
// Конструирует объект Stonewt из значения типа double
Stonewt::Stonewt(double lbs)
{
    stone = int (lbs) / Lbs_per_stn;  // целочисленное деление
    pds_left = int (lbs) % Lbs_per_stn + lbs - int(lbs);
    pounds = lbs;
}
```

```
// Конструирует объект Stonewt из стоунов и значения типа double
Stonewt::Stonewt(int stn, double lbs)
{
    stone = stn;
    pds_left = lbs;
    pounds = stn * Lbs_per_stn +lbs;
}
Stonewt::Stonewt()    // конструктор по умолчанию, wt =0
{
    stone = pounds = pds_left = 0;
}
Stonewt::~Stonewt()   // деструктор
{
}
// Отображение веса в стоунах
void Stonewt::show_stn() const
{
    cout << stone << " stone, " << pds_left << " pounds\n";
}
// Отображение веса в фунтах
void Stonewt::show_lbs() const
{
    cout << pounds << " pounds\n";
}
// Функции преобразования
Stonewt::operator int() const
{
    return int (pounds + 0.5);
}
Stonewt::operator double()const
{
    return pounds;
}
```

Код в листинге 11.21 тестирует новые функции преобразования. В операторе присваивания используется неявное преобразование, тогда как в последнем операторе `cout` применяется явное приведение типа. Обеспечьте совместную компиляцию кода в листингах 11.20 и 11.21.

Листинг 11.21. `stone1.cpp`

```
// stone1.cpp -- определяемые пользователем функции преобразования
// компилировать вместе с stonewt1.cpp
#include <iostream>
#include "stonewt1.h"
int main()
{
    using std::cout;
    Stonewt poppins(9,2.8);     // 9 стоунов, 2.8 фунта
    double p_wt = poppins;      // неявное преобразование
    cout << "Convert to double => ";
    cout << "Poppins: " << p_wt << " pounds.\n";
    cout << "Convert to int => ";
    cout << "Poppins: " << int (poppins) << " pounds.\n";
    return 0;
}
```

Ниже показан вывод программы из листингов 11.19, 11.20 и 11.21, который показывает результат преобразования объекта Stonewt в типы int и double:

```
Convert to double => Poppins: 128.8 pounds.
Convert to int => Poppins: 129 pounds.
```

Автоматическое применение преобразования типов

В листинге 11.21 используется int(poppins) с cout. Предположим, что явное приведение типов опущено:

```
cout << "Poppins: " << poppins << " pounds.\n";
```

Будет ли программа использовать неявное преобразование, как в следующем операторе?

```
double p_wt = poppins;
```

Ответ: нет, не будет. В примере с p_wt контекст указывает на то, что poppins должно быть приведено к типу double. Но в примере с cout ничего не говорит о том, должно ли выполняться приведение к int или double. Столкнувшись с такой нехваткой информации, компилятор предполагает, что вы применяете неоднозначное преобразование. Ничего в этом операторе не указывает на то, какой тип использовать.

Интересно, что если бы в классе была определена только одна функция приведения к double, в этом случае компилятор нормально бы принял показанный оператор. Причина в том, что если доступна только одна функция преобразования, то никакой двусмысленности нет.

Та же ситуация возникает и с присваиванием. При существующем объявлении класса компилятор отклоняет следующий оператор как неоднозначный:

```
long gone = poppins;            // неоднозначность
```

В C++ переменной типа long можно присваивать значения как int, так и double, поэтому компилятор на законном основании может использовать обе функции преобразования. Тем не менее, компилятор не является ответственным за выбор конкретной функции. Однако если исключить одну из этих двух функций, то компилятор обработает показанный оператор. Например, допустим, что удалено определение double(). В этом случае в процессе присваивания gone компилятор преобразует значение int в long.

Когда в классе определено два или более преобразований, вы по-прежнему можете использовать явное приведение типов, чтобы указать, какую именно функцию преобразования применять в конкретном случае. Можно использовать любую из следующих нотаций приведения:

```
long gone = (double) poppins;  // использовать преобразование в double
long gone = int (poppins);     // использовать преобразование в int
```

Первый из этих операторов преобразует вес poppins в значение типа double, а второй преобразует значение double в long.

Как и преобразующие конструкторы, функции преобразования могут успешно применяться в смешанном виде. Проблема с предоставлением функций, которые выполняют автоматическое неявное преобразование, состоит в том, что такое преобразование может происходить тогда, когда вы не ожидаете этого. Предположим, например, что вам по недоразумению случилось написать следующий код:

```
int ar[20];
...
Stonewt temp(14, 4);
```

```
...
int Temp = 1;
...
cout << ar[temp] << "!\n";  // непреднамеренное использование temp вместо
Temp
```

Обычно вы рассчитываете, что компилятор перехватит такие ошибки, как использование объекта вместо целого числа в индексе массива. Но в классе `Stonewt` определена функция `operator int()`, поэтому объект `temp` преобразуется в `int`-значение `200` и может применяться в качестве индекса массива. Мораль в том, что часто лучше использовать явные преобразования, запретив неявные. В C++98 ключевое слово `explicit` не работает с функциями преобразования, но в C++11 это ограничение снято. Таким образом, в C++11 операцию преобразования можно объявить как `explicit`:

```
class Stonewt
{
   ...
   // Функции преобразования
   explicit operator int() const;
   explicit operator double() const;
};
```

При наличии таких объявлений использование приведения типа приводит к вызову этих операций.

Другой подход состоит в том, чтобы заменить преобразующую функцию непреобразующей, которая решает ту же задачу, но только при явном вызове. То есть следующий код:

```
Stonewt::operator int() { return int (pounds + 0.5); }
```

можно заменить таким:

```
int Stonewt::Stone_to_Int() { return int (pounds + 0.5); }
```

Это запретит применение показанного ниже присваивания:

```
int plb = poppins;
```

Но если действительно нужно преобразование, будет разрешен следующий код:

```
int plb = poppins.Stone_to_Int();
```

Внимание!

Функции неявного преобразования следует применять осторожно. Часто лучшим выбором будет функция, которая может вызываться только явно.

Подведем некоторые итоги. В C++ для классов доступны перечисленные ниже преобразования типов.

- Конструктор класса, имеющий один аргумент, служит инструкцией по преобразованию значения типа аргумента в тип класса. Например, конструктор класса `Stonewt` с аргументом типа `int` вызывается автоматически, когда значение типа `int` присваивается объекту `Stonewt`. Однако использование ключевого слова `explicit` в объявлении конструктора исключает неявные преобразования и разрешает только явные.
- Специальная функция-член операции, называемая *функцией преобразования*, служит инструкцией для преобразования объекта класса в другой тип. Функция

преобразования является членом класса, не имеет типа возврата, не принимает аргументов и носит имя `operator` *имяТипа*`()`, где *имяТипа* — тип, в который преобразуется объект. Эта функция преобразования вызывается автоматически, когда вы присваиваете объект класса переменной соответствующего типа или применяете операцию приведения к этому типу.

Преобразования и друзья

Давайте добавим в класс `Stonewt` операцию сложения. Как упоминалось при обсуждении класса `Time`, для перегрузки операции сложения можно использовать либо функцию-член, либо дружественную функцию. (Для простоты предположим, что никаких функций преобразования в форме `operator double()` не определено.) Реализовать сложение можно с помощью следующей функции-члена:

```
Stonewt Stonewt::operator+(const Stonewt & st) const
{
   double pds = pounds + st.pounds;
   Stonewt sum(pds);
   return sum;
}
```

Либо же можно реализовать сложение в виде дружественной функции:

```
Stonewt operator+(const Stonewt & st1, const Stonewt & st2)
{
   double pds = st1.pounds + st2.pounds;
   Stonewt sum(pds);
   return sum;
}
```

Не забывайте, что можно предоставить либо определение метода, либо определение дружественной функции, но не оба сразу. Любая из этих форм делает возможным приведенный ниже код:

```
Stonewt jennySt(9, 12);
Stonewt bennySt(12, 8);
Stonewt total;
total = jennySt + bennySt;
```

Также в случае определения конструктора `Stonewt(double)` каждая из форм позволит применять следующие операторы:

```
Stonewt jennySt(9, 12);
double kennyD = 176.0;
Stonewt total;
total = jennySt + kennyD;
```

Но только дружественная функция разрешит такое:

```
Stonewt jennySt(9, 12);
double pennyD = 146.0;
Stonewt total;
total = pennyD + jennySt;
```

Чтобы увидеть, почему это так, необходимо транслировать каждое сложение в соответствующий вызов функции. Для начала, код

```
total = jennySt + bennySt;
```

транслируется в

```
total = jennySt.operator+(bennySt);          // функция-член
```

либо иначе:

```
total = operator+(jennySt, bennySt);         // дружественная функция
```

В каждом случае типы действительных аргументов соответствуют типам формальных аргументов. Кроме того, функция-член вызывается, как это и требуется, объектом `Stonewt`.

Далее оператор

```
total = jennySt + kennyD;
```

принимает следующий вид:

```
total = jennySt.operator+(kennyD);           // функция-член
```

либо иначе:

```
total = operator+(jennySt, kennyD);          // дружественная функция
```

Здесь снова функция-член вызывается, как положено — объектом `Stonewt`. На этот раз в каждом случае один аргумент (`kennyD`) имеет тип `double`, что вызывает конструктор `Stonewt(double)` для преобразования `double` в объект `Stonewt`.

Кстати, наличие функции-члена `operator double()` может здесь привести к путанице, поскольку создает дополнительный вариант для интерпретации. Вместо преобразования `kennyD` в `double` и сложения с `Stonewt` компилятор может преобразовать `jennySt` в `double` и выполнить сложение с `double`. Наличие слишком большого количества функций преобразования ведет к неоднозначности.

Наконец, оператор:

```
total = pennyD + jennySt;
```

превращается в

```
total = operator+(pennyD, jennySt);        // дружественная функция
```

Здесь оба аргумента имеют тип `double`, что вызывает конструктор `Stonewt(double)` для преобразования их в объекты `Stonewt`.

Однако версия с функцией-членом не сможет сложить `jennySt` с `pennyD`. После трансляции в вызов функции синтаксис сложения будет выглядеть следующим образом:

```
total = pennyD.operator+(jennySt);         // бессмыслица
```

Но этот код не имеет смысла, поскольку только объект класса может вызывать функцию-член. Компилятор не пытается преобразовать `pennyD` в объект `Stonewt`. Преобразование имеет место только для аргументов функций-членов, а не для объектов, на которых они вызываются.

Из сказанного можно сделать вывод: определение сложения как дружественной функции упрощает для программы задачу автоматических преобразований типов. Причина в том, что оба операнда становятся аргументами функции, поэтому прототипирование функции применяется в отношении обоих операндов.

Выбор реализации сложения

Предположим, что вы хотите складывать величины типа `double` и `Stonewt`. При этом доступно несколько вариантов. Первый, как вы уже видели, состоит в том, чтобы определить следующую операцию в виде дружественной функции и иметь конст-

руктор Stonewt(double), выполняющий преобразование аргументов типа double в аргументы типа Stonewt:

```
operator+(const Stonewt &, const Stonewt &)
```

Второй вариант — использовать для перегрузки операции сложения функцию, которая явно принимает один аргумент типа double:

```
Stonewt operator+(double x);        // функция-член
friend Stonewt operator+(double x, Stonewt & s);
```

В этом случае приведенный ниже оператор явно соответствует функции-члену operator+(double x):

```
total = jennySt + kennyD;           // Stonewt + double
```

А следующий оператор явно соответствует функции-члену operator+(double x, Stonewt & s):

```
total = pennyD + jennySt;           // double + Stonewt
```

Ранее нечто подобное делалось для умножения в классе Vector.

Каждый вариант имеет свои преимущества. Первый (полагающийся на неявные преобразования) в результате дает более короткие программы, поскольку определяется меньше функций. Это также означает меньший объем работы и уменьшение количества потенциальных ошибок. Недостаток связан с дополнительными накладными расходами времени и памяти на вызов преобразующих конструкторов каждый раз, когда требуется преобразование. Второй вариант (с дополнительными функциями, явно соответствующими типам) является зеркальным отражением первого. Он приводит к удлинению программ и требует дополнительных усилий при реализации, но работает несколько быстрее.

Если в программе интенсивно применяется сложение значений double с объектами Stonewt, может быть, стоит перегрузить операцию сложения для эффективной реализации этой операции. Если же программа применяет такое сложение изредка, то проще положиться на автоматическое преобразование типов либо, если вы хотите быть более аккуратными — на явное преобразование.

Резюме

В настоящей главе было раскрыто много важных аспектов определения и использования классов. Некоторые из материалов главы могут показаться неясными до тех пор, пока вы не углубите понимание на основе собственного опыта.

Обычно единственным способом доступа к закрытым членам класса является использование методов класса. В C++ это ограничение ослабляется за счет технологии дружественных функций. Чтобы сделать функцию другом класса, ее необходимо объявить внутри объявления класса и предварить объявление ключевым словом friend.

C++ расширяет перегрузку операций возможностью определять специальные функции операций, которые описывают, как конкретная операция применяется с конкретным классом. Функция операции может быть функцией-членом класса либо дружественной функцией. (Некоторые операции могут быть только членами класса.) C++ позволяет вызывать функцию операции как непосредственным обращением к этой функции, так и применением операции в обычном синтаксисе. Функция операции для операции *op* имеет следующую форму:

operator*op*(*список-аргументов*)

список-аргументов представляет операнды операции. Если функция операции является функцией-членом класса, то первый операнд — это вызывающий объект, не являющийся частью *список-аргументов*. Например, в настоящей главе вы перегружали операцию сложения, определив функцию `operator+()` в классе `Vector`. Если `up`, `right` и `result` — три вектора, для сложения векторов можно использовать любой из следующих операторов:

```
result = up.operator+(right);
result = up + right;
```

Во втором варианте тот факт, что операнды `up` и `right` имеют тип `Vector`, заставляет C++ применять определение сложения, объявленное в классе `Vector`.

Когда функция операции является функцией-членом, первый операнд представляет собой объект, вызывающий эту функцию. Так, например, в приведенных выражениях объект `up` является вызывающим объектом. Если хотите определить функцию операции так, чтобы первый операнд не был объектом класса, необходимо объявить ее как дружественную функцию. Тогда ей можно будет передавать операнды в любом порядке.

Одна из наиболее часто применяемых задач перегрузки — определение операции `<<` для ее применения с объектом `cout` с целью отображения содержимого объектов. Чтобы позволить объекту `ostream` быть первым операндом, функция операции определяется как дружественная. Чтобы позволить перегруженной операции сцепляться с самой собой, в качестве возвращаемого типа этой функции указывается `ostream &`. Ниже приведена общая форма, удовлетворяющая этим требованиям:

```
ostream & operator<<(ostream & os, const c_name & obj)
{
   os << ... ;            // отображение содержимого объекта
   return os;
}
```

Однако если класс имеет методы, возвращающие значения данных-членов, которые нужно отобразить, их можно использовать в `operator<<()` вместо прямого доступа. В этом случае функция не обязана быть дружественной.

C++ позволяет устанавливать преобразования типов из класса в заданный тип и обратно. Прежде всего, любой конструктор класса, принимающий единственный аргумент, может служить функцией преобразования, преобразуя тип аргумента в тип класса. Конструктор вызывается автоматически, когда значение типа аргумента присваивается объекту. Например, предположим, что имеется класс `String` с конструктором, который принимает один аргумент типа `char *`. Тогда если `bean` — объект типа `String`, то можно записать следующий оператор:

```
bean = "pinto";          // преобразует тип char * в тип String
```

Если объявлению конструктора предшествует ключевое слово `explicit`, то конструктор может быть использован только для явного преобразования:

```
bean = String("pinto");  // преобразует тип char * в тип String явным образом
```

Для преобразования из класса в другой тип потребуется определить функцию преобразования и предоставить инструкции о том, как выполнять это преобразование. Функция преобразования должна быть функцией-членом. Если она преобразует в тип *имяТипа*, то ее прототип должен выглядеть следующим образом:

```
operator имяТипа();
```

Обратите внимание, что функция не имеет объявленного типа возврата, не принимает аргументов и возвращает преобразованное значение (несмотря на отсутствие возвращаемого типа). Например, функция для преобразования типа `Vector` в тип `double` должна иметь такую форму:

```
Vector::operator double()
{
   ...
   return значение_double;
}
```

Опыт показывает, что часто лучше не полагаться на функции неявного преобразования.

Возможно, вы уже отметили, что классы требуют гораздо более пристального внимания к деталям, нежели простые структуры в стиле C. Однако взамен они приносят гораздо бо́льшую пользу.

Вопросы для самоконтроля

1. Воспользуйтесь функцией-членом для перегрузки операции умножения в классе `Stonewt`; определите операцию умножения членов данных на значение типа `double`. Имейте в виду, что нужно будет позаботиться о представлении "стоун-фунт". То есть удвоение 10 стоунов и 8 фунтов должно давать 21 стоун и 2 фунта.
2. В чем отличия между дружественной функцией и функцией-членом?
3. Должна ли функция, не являющаяся членом, быть дружественной для того, чтобы иметь доступ к членам класса?
4. Воспользуйтесь дружественной функцией для перегрузки операции умножения в классе `Stonewt`; определите операцию умножения значения `double` на значение `Stone`.
5. Какие операции не могут быть перегружены?
6. Какие ограничения накладываются на перегрузку следующих операций: `=`, `()`, `[]` и `->`?
7. Определите функцию преобразования для класса `Vector`, которая будет приводить объект `Vector` к значению типа `double`, которое представляет длину вектора.

Упражнения по программированию

1. Модифицируйте код в листинге 11.15 так, чтобы обеспечить запись в файл последовательных позиций при случайном блуждании. Каждая позиция должна помечаться номером шага. Также программа должна записывать в файл начальные условия (целевое расстояние и длину шага) и суммарные результаты. Содержимое файла может выглядеть примерно так:

```
Target Distance: 100, Step Size: 20
0: (x,y) = (0, 0)
1: (x,y) = (-11.4715, 16.383)
2: (x,y) = (-8.68807, -3.42232)
...
26: (x,y) = (42.2919, -78.2594)
27: (x,y) = (58.6749, -89.7309)
After 27 steps, the subject has the following location:
```

```
(x,y) = (58.6749, -89.7309)
 or
(m,a) = (107.212, -56.8194)
Average outward distance per step = 3.97081
```

2. Модифицируйте заголовок класса `Vector` и файлы реализации (листинги 11.13 и 11.14) так, чтобы модуль и направление вектора больше не хранились в виде компонентов данных. Вместо этого они должны вычисляться по требованию при вызове методов `magval()` и `angval()`. Вы должны оставить открытый интерфейс без изменений (те же открытые методы с теми же аргументами), но изменить закрытую часть, включая некоторые из закрытых методов и их реализации. Протестируйте модифицированную версию с помощью программы из листинга 11.15, которая должна остаться неизменной, поскольку открытый интерфейс класса `Vector` не менялся.

3. Модифицируйте код в листинге 11.15 так, чтобы вместо сообщений о результатах одиночной попытки при конкретной комбинации расстояние/шаг сообщалось максимальное, минимальное и среднее количество шагов для N попыток, где N — целое число, вводимое пользователем.

4. Перепишите финальный пример класса `Time` (листинги 11.10, 11.11 и 11.12) так, чтобы все перегруженные операции были реализованы с использованием дружественных функций.

5. Перепишите класс `Stonewt` (листинги 11.16 и 11.17) так, чтобы он имел член состояния, который управляет тем, в какой форме интерпретируется объект: стоуны, целочисленное значение в фунтах или значение в фунтах с плавающей точкой. Перегрузите операцию << для замены методов `show_stn()` и `show_lbs()`. Перегрузите операции сложения, вычитания и умножения значений `Stonewt`. Протестируйте полученный класс с помощью короткой программы, в которой используются все методы и друзья класса.

6. Перепишите класс `Stonewt` (листинги 11.16 и 11.17) так, чтобы перегружались все шесть операций сравнения. Операции должны сравнивать члены `pounds` и возвращать значение типа `bool`. Напишите программу, которая объявляет массив из шести объектов `Stonewt` с инициализацией в объявлении первых трех из них. Затем программа должна в цикле читать значения, используемые для установки остальных трех элементов массива. После этого программа должна вывести самый маленький элемент, самый большой, а также количество элементов, которые больше или равны 11 стоунам. (Простейший подход предполагает создание объекта `Stonewt`, инициализированного 11 стоунами, и сравнение с ним других объектов.)

7. Комплексное число состоит из двух частей — вещественной и мнимой. Один из способов записи такого числа выглядит как `(3.0, 4.0)`. Здесь `3.0` — вещественная часть, а `4.0` — мнимая. Предположим, что a = (A, Bi) и c = (C, Di). Ниже представлены некоторые операции с комплексными числами:
   - сложение: a + c = (A + C, (B + D)i)
   - вычитание: a – c = (A – C, (B – D)i)
   - умножение: a*c = (A×C – B×D, (A×D + B×C)i)
   - умножение (x — вещественное число): x×c = (x×C, x×Di)
   - сопряжение: ~a = (A, – Bi)

Определите класс `complex` так, чтобы следующая программа могла использовать его с корректными результатами:

```
#include <iostream>
using namespace std;
#include "complex0.h"     // во избежание конфликта с complex.h
int main()
{
   complex a(3.0, 4.0);  // инициализация значением (3,4i)
   complex c;
   cout << "Enter a complex number (q to quit):\n";
         // Ввод комплексного числа (q для завершения)
   while (cin >> c)
   {
      cout << "c is " << c << '\n';              // значение c
      cout << "complex conjugate is " << ~c << '\n';
           // значение сопряженного числа
      cout << "a is " << a << '\n';              // значение a
      cout << "a + c is " << a + c << '\n';      // значение a + c
      cout << "a - c is " << a - c << '\n';      // значение a - c
      cout << "a * c is " << a * c << '\n';      // значение a * c
      cout << "2 * c is " << 2 * c << '\n';      // значение 2 * c
      cout << "Enter a complex number (q to quit):\n";
   }
   cout << "Done!\n";
   return 0;
}
```

Не забывайте, что вы должны перегрузить операции << и >>. В стандарте C++ уже присутствует поддержка комплексных чисел — и намного более развитая, чем в этом примере — в заголовочном файле `complex`, поэтому во избежание конфликтов назовите свой файл `complex0.h`. Используйте `const` там, где это оправдано.

Ниже показан пример выполнения этой программы:

```
Enter a complex number (q to quit):
real: 10
imaginary: 12
c is (10,12i)
complex conjugate is (10,-12i)
a is (3,4i)
a + c is (13,16i)
a - c is (-7,-8i)
a * c is (-18,76i)
2 * c is (20,24i)
Enter a complex number (q to quit):
real: q
Done!
```

Обратите внимание, что благодаря перегрузке, `cin >> c` теперь запрашивает ввод вещественной и мнимой частей комплексного числа.

12

Классы и динамическое выделение памяти

В ЭТОЙ ГЛАВЕ...

- Динамическое выделение памяти для членов класса
- Явные и неявные конструкторы копирования
- Явные и неявные перегруженные операции присваивания
- Что необходимо делать при использовании операции `new` в конструкторе
- Использование статических членов класса
- Создание объектов операцией `new` с размещением
- Использование указателей на объекты
- Реализация абстрактного типа данных очереди

В данной главе описывается применение операций new и delete с классами, а также некоторые тонкости, которые следует учитывать при использовании динамической памяти. Вроде бы совсем немного вопросов, однако, эти вопросы затрагивают и разработку конструкторов, и разработку деструкторов, и перегрузку операций.

Рассмотрим конкретный пример, показывающий, как C++ помогает управлять загрузкой памяти. Допустим, что нужно создать класс с членом, представляющим чью-либо фамилию. Самым простым способом для хранения имени является использование символьного массива. Однако этот способ имеет несколько недостатков. Предположим, что для фамилии выделен 14-символьный массив — и тут на сцене появляется некто с фамилией из трех десятков букв. Конечно, надежнее использовать 40-символьный массив. Однако если создать массив из 2000 подобных объектов, то огромный объем памяти в частично заполненных элементах будет не задействован, а заодно увеличивается расход и оперативной памяти компьютера. Но возможен и другой вариант.

Некоторые вопросы (например, сколько памяти использовать) часто удобнее решать во время выполнения программы, а не ее компиляции. Для хранения имени в объекте в C++ обычно применяется операция new в конструкторе класса — что позволяет выделить нужный объем памяти при работе программы. Как правило, это делается с помощью класса string, который берет на себя все заботы об управлении памятью. Но так вы ничего не узнаете об управлении памятью, поэтому мы будем решать задачу напрямую. Применение операции new в конструкторе класса приводит к появлению нескольких новых проблем, для решения которых приходится предпринимать дополнительные меры: расширение класса деструктора, согласование всех конструкторов с деструктором new и написание дополнительных методов класса для обеспечения правильности инициализации и присваивания. (Конечно, в этой главе будут объяснены все эти шаги.)

Динамическая память и классы

Что вы предпочтете на завтрак, обед и ужин в следующем месяце? Сколько чашек молока в обед третьего числа? А сколько изюминок в каше на завтрак пятнадцатого числа? Большинство людей откладывает принятие таких решений непосредственно до момента приема пищи. C++ аналогично относится к распределению памяти: пусть программа принимает решения относительно памяти во время выполнения, а не во время компиляции. Тогда потребление памяти будет зависеть только от потребностей программы, а не от набора жестких правил для классов памяти. Вспомните, что для динамического управления памятью в C++ используются операции new и delete. К сожалению, применение данных операций с классами может породить новые проблемы. Как мы увидим, деструкторы из просто декоративных элементов могут стать жизненно необходимыми. Иногда даже понадобится перегружать операцию присваивания, чтобы программа вела себя должным образом. Этими вопросами мы сейчас и займемся.

Простой пример и статические члены класса

Мы уже давно не пользовались операциями new и delete, поэтому давайте рассмотрим их на примере короткой программы. А заодно познакомимся и с новым классом хранения — статическим членом класса. Сначала это будет класс StringBad, который впоследствии мы заменим несколько более функциональным классом String. (Вы уже знакомы со стандартным классом C++ string, и еще продолжите знакомство в главе 16. Но простые классы StringBad и String, обсуждаемые в данной главе, по-

могут понять принципы работы подобных классов. Множество программных технологий ориентируются на предоставление такого дружественного интерфейса.)

Объекты классов `StringBad` и `String` содержат указатель на строку и значение, представляющее длину строки. Мы используем эти классы в основном для того, чтобы уяснить механизм работы `new`, `delete` и членов классов `static`. Поэтому конструкторы и деструкторы во время вызова будут выводить сообщения, помогающие проследить их работу. Зато для упрощения интерфейса класса мы опустим некоторые полезные функции-члены и дружественные функции (вроде перегруженных операций ++ и >>) и функцию преобразования. (Не переживайте, вопросы для самоконтроля в конце данной главы предоставят вам возможность добавить эти полезные функции.) Объявление класса приведено в листинге 12.1.

Листинг 12.1. `strngbad.h`

```
// strngbad.h -- несовершенное определение класса строки
#include <iostream>
#ifndef STRNGBAD_H_
#define STRNGBAD_H_
class StringBad
{
private:
   char * str;                    // указатель на строку
   int len;                       // длина строки
   static int num_strings;        // количество объектов
public:
   StringBad(const char * s);     // конструктор
   StringBad();                   // конструктор по умолчанию
   ~StringBad();                  // деструктор

   // Дружественная функция
   friend std::ostream & operator<<(std::ostream & os, const StringBad & st);
};
#endif
```

Класс называется `StringBad` — в качестве напоминания о том, что это пример незаконченной разработки. Сейчас разрабатывается класс с использованием распределения динамической памяти, и очевидные действия класс выполняет корректно. Например, в нем правильно используются операции `new` и `delete` в конструкторах и деструкторах. В принципе, в нем нет ничего "нехорошего", просто он не содержит реализацию некоторых дополнительных "хороших" вещей, которые необходимы, но совсем не очевидны. Когда мы уясним проблемы, связанные с этим классом, мы сможем понять и запомнить те неочевидные изменения, которые нужно будет сделать впоследствии, при преобразовании его в более функциональный класс `String`.

Обратите внимание на два момента в этом объявлении. Во-первых, в нем для представления фамилии используется указатель на `char`, а не массив `char`. Это означает, что объявление класса не выделяет непосредственно память для строки. Для этого применяется операция `new` в конструкторах. Такая схема позволяет избежать привязки объявления класса к предопределенной границе размера строки.

Во-вторых, в определении член `num_strings` объявлен как принадлежащий к классу хранения `static`. *Статический член класса* обладает особым свойством: программа создает только одну копию статической переменной класса независимо от количества создаваемых объектов. Другими словами, статический член совместно используется всеми объектами данного класса — как единый номер телефона для всех членов семьи.

Если, к примеру, создано десять объектов StringBad, то в них будут содержаться десять членов str и десять членов len, но лишь один общий член num_strings (рис. 12.1). Это удобно для данных, которые являются закрытыми для класса, но при этом должны иметь одно общее значение для всех объектов класса. Например, член num_strings предназначен для отслеживания количества создаваемых объектов.

```
Class StringBad
{
private:
   char * str;
   int len;
   static int num_strings;
public:
   ...
};
```

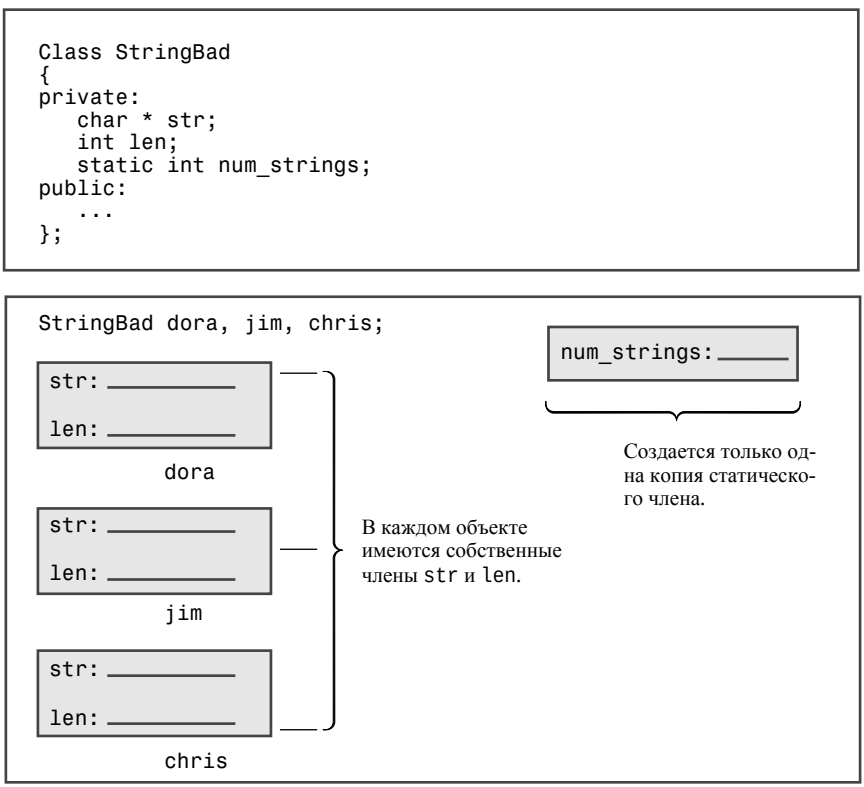


Рис. 12.1. *Статические данные-члены*

Кстати, в листинге 12.1 член num_strings служит просто для демонстрации статических данных-членов, а также для того, чтобы обратить внимание на возможные проблемы программирования. Вообще говоря, в строковом классе такой член не нужен.

Взгляните на реализацию методов класса в листинге 12.2. Обратите внимание на то, как в нем используется указатель и статический член.

Листинг 12.2. strngbad.cpp

```
// strngbad.cpp -- методы класса StringBad
#include <cstring>    // в некоторых случаях — string.h
#include "strngbad.h"
using std::cout;

// Инициализация статического члена класса
int StringBad::num_strings = 0;

// Методы класса

// Создание StringBad из C-строки
StringBad::StringBad(const char * s)
{
    len = std::strlen(s);                          // установка размера
    str = new char[len + 1];                       // выделение памяти
    std::strcpy(str, s);                           // инициализация указателя
    num_strings++;                                 // счетчик объектов
    cout << num_strings << ": \"" << str
         << "\" object created\n";                 // для целей отладки
}

StringBad::StringBad()                             // конструктор по умолчанию
{
    len = 4;
    str = new char[4];
    std::strcpy(str, "C++");                       // строка по умолчанию
    num_strings++;
    cout << num_strings << ": \"" << str
         << "\" default object created\n";         // для целей отладки
}

StringBad::~StringBad()                            // необходимый деструктор
{
    cout << "\"" << str << "\" object deleted, ";  // для целей отладки
    --num_strings;                                 // является обязательным
    cout << num_strings << " left\n";              // для целей отладки
    delete [] str;                                 // является обязательным
}

std::ostream & operator<<(std::ostream & os, const StringBad & st)
{
    os << st.str;
    return os;
}
```

Прежде всего, обратите внимание на следующий оператор в листинге 12.2:

```
int StringBad;:num_strings = 0;
```

Этот оператор устанавливает первоначальное значение 0 для статического члена num_strings. Учтите, что статическую переменную-член нельзя инициализировать внутри объявления класса. Ведь объявление — это описание того, как выделяется память, но не само выделение. Память выделяется и инициализируется при создании объекта на основе данного формата. Для статического члена класса осуществляется независимая инициализация — с помощью отдельного оператора вне объявления класса. Это объясняется тем, что статический член класса хранится не в составе объектов. Обратите внимание, что оператор инициализации задает тип и указывает область действия, но не содержит ключевое слово static.

Такая инициализация записывается в файле методов, а не в файле объявления класса, поскольку объявление класса содержится в заголовочном файле. Причина в том, что заголовочный файл может быть включен в несколько других файлов программы, и тогда оператор инициализации будет ошибочно выполнен несколько раз.

Существует исключение, когда статические данные-члены все-таки инициализируются внутри объявления класса (см. главу 10) — если статический член данных определяется как константа целочисленного или перечислимого типа.

На заметку!

Статические данные-члены объявляются в объявлении класса и инициализируются в файле, содержащем методы класса. При инициализации используется операция разрешения контекста, чтобы указать, к какому классу принадлежит статический член. Однако если статический член определен как `const` целочисленного или перечислимого типа, то его можно инициализировать непосредственно в объявлении класса.

Каждый конструктор содержит выражение `num_strings++`. Это значит, что каждый раз, когда программа создает новый объект, общая переменная `num_strings` увеличивается на единицу, т.е. всегда содержит общее количество объектов `String`. А деструктор содержит выражение `--num_strings`.

Таким образом, класс `String` отслеживает не только создание, но и удаление объектов, храня в переменной `num_strings` их текущее количество.

Теперь рассмотрим первый конструктор из листинга 12.2, который инициализирует объект `String` обычной строкой в стиле C:

```
StringBad::StringBad(const char * s)
{
    len = std::strlen(s);                          // установка размера
    str = new char[len + 1];                       // выделение памяти
    std::strcpy(str, s);                           // инициализация указателя
    num_strings++;                                 // счетчик объектов
    cout << num_strings << ": \"" << str
         << "\" object created\n";                 // для целей отладки
}
```

Член класса `str` — это просто указатель, поэтому конструктор должен предоставить память для хранения строки. Указатель строки можно передать конструктору при инициализации объекта:

```
String boston("Boston");
```

Затем конструктор должен выделить объем памяти, достаточный для хранения строки, и скопировать строку в это место. Давайте проследим этот процесс более подробно.

Сначала функция инициализирует член `len`, используя функцию `strlen()` для вычисления длины строки. После этого применяется операция `new`, чтобы выделить память, достаточную для хранения строки, и адрес этого участка памяти заносится в член `str`. (Поскольку функция `strlen()` возвращает длину строки без завершающего нулевого символа, конструктор увеличивает `len` на единицу, чтобы уместить строку с нулевым символом.)

После этого в конструкторе используется функция `strcpy()` для копирования передаваемой строки в новый участок памяти. Затем обновляется счетчик объектов. И в завершение, чтобы можно было следить за происходящим, конструктор выводит текущее количество объектов и строку, хранящуюся в объекте. Эта возможность пригодится позднее, когда мы будем загонять класс `String` в разные неприятности.

Чтобы понять данный принцип, вы должны осознавать, что строка не хранится в объекте. Символы находятся отдельно, в куче, а сам объект просто указывает, где их найти.

Учтите, что так делать нельзя:

```
str = s; // не делайте так
```

Этот код просто сохраняет адрес, но не создает копию строки.

Конструктор по умолчанию работает аналогично, только он заносит в строку значение по умолчанию "C++".

Деструктор в этом примере содержит самое важное дополнение к обработке классов:

```
StringBad::~StringBad()                          // необходимый деструктор
{
   cout << "\"" << str << "\" object deleted, "; // для целей отладки
   --num_strings;                                // является обязательным
   cout << num_strings << " left\n";             // для целей отладки
   delete [] str;                                // является обязательным
}
```

Деструктор начинает свою работу с уведомления, что он вызван. Эта часть кода информативна, но не обязательна. А вот операция `delete` необходима. Ведь член `str` указывает на память, выделенную операцией `new`. При уничтожении объекта `StringBad` исчезает и указатель `str`. Однако память, на которую указывал `str`, остается выделенной, пока не будет освобождена операцией `delete`. Удаление объекта освобождает память, занимаемую самим объектом, но при этом не освобождается автоматически память, адресованная указателями, которые были членами объекта. Для этого нужно использовать деструктор. Операция `delete` в деструкторе перед удалением объекта освобождает память, выделенную в конструкторе операцией `new`.

На заметку!

Всякий раз, когда в конструкторе для выделения памяти используется операция `new`, в соответствующем деструкторе необходима операция `delete` для освобождения этой памяти. Если использовалась операция `new []` (с квадратными скобками), то нужно применять операцию `delete []` (тоже с квадратными скобками).

В листинге 12.3 демонстрируется работа конструкторов и деструкторов `StringBad`. В программе объявления объектов размещены во внутреннем блоке, т.к. деструктор вызывается тогда, когда управление покидает блок, в котором определен объект. Без внутреннего блока деструкторы были бы вызваны после выхода из программы `main()` — в некоторых средах это не позволит увидеть сообщения деструкторов, прежде чем будет закрыто окно программы. Не забудьте компилировать вместе с кодом из листинга 12.3 и код из листинга 12.2.

Листинг 12.3. vegnews.cpp

```
// vegnews.cpp -- использование операций new и delete с классами
// компилировать вместе с strngbad.cpp
#include <iostream>
using std::cout;
#include "strngbad.h"

void callmel(StringBad &);  // передача по ссылке
void callme2(StringBad);    // передача по значению
```

```
int main()
{
   using std::endl;
   {
      cout << "Starting an inner block.\n";
      StringBad headline1("Celery Stalks at Midnight");
      StringBad headline2("Lettuce Prey");
      StringBad sports("Spinach Leaves Bowl for Dollars");
      cout << "headline1: " << headline1 << endl;
      cout << "headline2: " << headline2 << endl;
      cout << "sports: " << sports << endl;
      callme1(headline1);
      cout << "headline1: " << headline1 << endl;
      callme2(headline2);
      cout << "headline2: " << headline2 << endl;
      cout << "Initialize one object to another:\n";
      StringBad sailor = sports;
      cout << "sailor: " << sailor << endl;
      cout << "Assign one object to another:\n";
      StringBad knot;
      knot = headline1;
      cout << "knot: " << knot << endl;
      cout << "Exiting the block.\n";
   }
   cout << "End of main()\n";
   return 0;
}
void callme1(StringBad & rsb)
{
   cout << "String passed by reference:\n";  // строка, переданная по ссылке
   cout << " \"" << rsb << "\"\n";
}
void callme2(StringBad sb)
{
   cout << "String passed by value:\n";     // строка, переданная по значению
   cout << " \"" << sb << "\"\n";
}
```

На заметку!

Данный черновой проект `StringBad` содержит несколько преднамеренно внесенных дефектов, из-за которых точный вид выходных данных не определен. Например, некоторые компиляторы генерируют исполняемый код, который завершается аварийно и преждевременно. Но хотя конкретный вид выходных данных может отличаться, основные проблемы и способы их устранения (которые вскоре будут описаны) одинаковы.

Ниже показан вывод, полученный программой из листинга 12.3, которая была скомпилирована компилятором командной строки Borland C++ 5.5:

```
Starting an inner block.
1: "Celery Stalks at Midnight" object created
2: "Lettuce Prey" object created
3: "Spinach Leaves Bowl for Dollars" object created
headline1: Celery Stalks at Midnight
headline2: Lettuce Prey
sports: Spinach Leaves Bowl for Dollars
```

```
String passed by reference:
   "Celery Stalks at Midnight"
headline1: Celery Stalks at Midnight
String passed by value:
   "Lettuce Prey"
"Lettuce Prey" object deleted, 2 left
headline2: Dû°
Initialize one object to another:
sailor: Spinach Leaves Bowl for Dollars
Assign one object to another:
3: "C++" default object created
knot: Celery Stalks at Midnight
Exiting the block.
"Celery Stalks at Midnight" object deleted, 2 left
"Spinach Leaves Bowl for Dollars" object deleted, 1 left
"Spinach Leaves Bowl for Doll8" object deleted, 0 left
"@g" object deleted, -1 left
"-|" object deleted, -2 left
End of main()
```

Различные нестандартные символы, которые появляются в выводе, будут отличаться в зависимости от системы; это одна из причин, по которой класс `StringBad` назван неудачным (часть `Bad` в имени). Другая причина — отрицательные значения счетчика объектов. Сочетания более новых компиляторов и операционных систем, как правило, приводят к аварийному завершению программы непосредственно перед выводом строки с сообщением, что остался -1 объект (`-1 left`), а некоторые выдают сообщение General Protection Fault (GPF, общее нарушение защиты). Появление GPF означает, что программа пытается получить доступ к запрещенному для нее месту в памяти — это еще один признак неудачного проекта.

Замечание по программе

Программа в листинге 12.3 начинает работать нормально, но потом идет к странному аварийному завершению. Рассмотрим сначала то, что работает нормально. Конструктор выдает сообщение о том, что он создал три объекта `StringBad`, и нумерует их. Программа выводит созданные объекты, используя перегруженную операцию <<:

```
Starting an inner block.
1: "Celery Stalks at Midnight" object created·
2: "Lettuce Prey" object created
3: "Spinach Leaves Bowl for Dollars" object created
headline1: Celery Stalks at Midnight
headline2: Lettuce Prey
sports: Spinach Leaves Bowl for Dollars
```

После этого программа передает переменную `headline1` в функцию `callme1()` и после вызова снова выводит содержимое `headline1`. Вот этот код:

```
callme1(headline1);
cout << "headline1: " << headline1 << endl;
```

А вот результат:

```
String passed by reference:
   "Celery Stalks at Midnight"
headline1: Celery Stalks at Midnight
```

Этот раздел кода тоже работает нормально.

Но потом в программе выполняется следующий фрагмент кода:

```
callme2(headline2);
cout << "headline2: " << headline2 << endl;
```

Здесь функция `callme2()` передает `headline2` по значению, а не по ссылке, и результат указывает на серьезную проблему:

```
String passed by value:
   "Lettuce Prey"
"Lettuce Prey" object deleted, 2 left
headline2: Dû°
```

Во-первых, при передаче `headline2` в качестве аргумента функции почему-то был вызван деструктор. Во-вторых, передача по значению должна защищать исходный аргумент от изменения, однако исходная строка изуродована так, что появились нестандартные символы. (Конкретный текст зависит от содержимого памяти в данный момент.)

Но самый кошмар находится в конце вывода, когда деструктор автоматически вызывается для каждого из созданных ранее объектов:

```
Exiting the block.
"Celery Stalks at Midnight" object deleted, 2 left
"Spinach Leaves Bowl for Dollars" object deleted, 1 left
"Spinach Leaves Bowl for Doll8" object deleted, 0 left
"@g" object deleted, -1 left
"-|" object deleted, -2 left
End of main()
```

Поскольку объекты с автоматическим хранением удаляются в порядке, обратном их созданию, то сначала удаляются объекты `knots`, `sailor` и `sport`. Удаления `knots` и `sailor` проходят нормально, но для `sport` вместо `Dollars` появляется `Doll8`. Единственное обращение программы к объекту `sport` — при инициализации `sailor`, но, похоже, что именно здесь `sport` и изменяется. А последние два удаленных объекта, `headline2` и `headline1`, искажены до неузнаваемости. Что-то запортило данные этих строк перед их удалением. Странно выглядит и счетчик: как может остаться –2 объекта?

Но этот странный подсчет как раз и дает ключ к разгадке. Каждый объект создается один раз и удаляется один раз — значит, количество вызовов конструктора должно равняться количеству вызовов деструктор. Поскольку счетчик объектов (`num_strings`) декрементируется на два раза больше, чем инкрементируется, то два объекта должен создать конструктор, который не инкрементирует значение `num_strings`. В описании класса объявлены и определены два конструктора (и оба инкрементируют `num_strings`), но оказывается, что в программе используются три конструктора. Рассмотрим, например, следующую строку:

```
StringBad sailor = sports;
```

Какой конструктор здесь используется? Не конструктор по умолчанию и не конструктор с параметром `const char *`. Вспомните, что инициализация, применяющая данную форму, должна иметь другой синтаксис:

```
StringBad sailor = StringBad(sports);  // конструктор, использующий sports
```

Поскольку объект `sports` имеет тип `StringBad`, соответствующий конструктор должен обладать следующим прототипом:

```
StringBad(const StringBad &);
```

Оказывается, что компилятор автоматически генерирует этот конструктор (называемый *конструктором копирования*, поскольку он создает копию объекта), если один объект инициализируется другим. Автоматически сгенерированной версии конструктора ничего не известно про обновление статической переменной `num_strings`, поэтому подсчет и нарушается. Так что все проблемы, обнаруженные в данном примере, возникли из-за функций-членов, которые компилятор генерирует автоматически. Давайте рассмотрим данную тему.

Специальные функции-члены

Проблемы с классом `StringBad` возникают из-за *специальных функций-членов*, которые определяются автоматически. В случае `StringBad` поведение этих функций-членов не соответствует конкретному построению класса. В частности, C++ автоматически предоставляет следующие функции-члены:

- конструктор по умолчанию, если не было определено ни одного конструктора;
- деструктор по умолчанию, если он не был определен;
- конструктор копирования, если он не был определен;
- операция присваивания, если она не была определена;
- операция взятия адреса, если она не была определена.

Точнее, компилятор генерирует определения для трех последних элементов, если программа использует объекты так, что эти определения будут нужны. Например, если где-то выполняется присваивание одного объекта другому, то программа предоставляет определение для операции присваивания.

Оказывается, причиной проблем с классом `StringBad` являются неявный конструктор копирования и неявная операция присваивания.

Неявная операция взятия адреса возвращает адрес вызывающего объекта (т.е. значение указателя `this`). Эта функция вполне годится для наших целей, поэтому мы не будем больше обсуждать ее. Деструктор по умолчанию ничего не делает, поэтому мы не будем рассматривать и его – просто запомним, что в классе уже имеется подмена для него. Однако остальное стоит рассмотреть более подробно.

В C++11 предлагаются еще две специальные функции-члена – *конструктор переноса* и *операция присваивания с переносом*. Они будут описаны в главе 18.

Конструкторы по умолчанию

Если для класса вообще не задан какой-либо конструктор, то C++ предоставляет конструктор по умолчанию. Пусть, например, определен класс `Klunk`, в котором нет конструкторов. В таком случае компилятор снабжает код следующим стандартным оператором:

```
Klunk::Klunk() { }          // неявный конструктор по умолчанию
```

Другими словами, он предоставляет конструктор (*стандартизированный* конструктор по умолчанию), который не принимает аргументы и вообще ничего не делает. Но он необходим, поскольку при создании объекта всегда вызывается конструктор:

```
Klunk lunk;                 // вызывает конструктор по умолчанию
```

Из-за конструктора по умолчанию переменная `lunk` выглядит как обычная автоматическая переменная: ее значение при инициализации неизвестно.

Если же определен хоть какой-нибудь конструктор, C++ не считает необходимым определять конструктор по умолчанию. Если требуется создавать объекты, которые

не инициализируются явно, то придется явно определить конструктор по умолчанию. Хотя это конструктор без аргументов, его можно применять для установки отдельных значений:

```
Klunk::Klunk()    // явный конструктор по умолчанию
{
   klunk_ct = 0;
   ...
}
```

Конструктор с аргументами по-прежнему может быть конструктором по умолчанию, если все его аргументы имеют значения по умолчанию. Например, класс `Klunk` может содержать следующий встроенный конструктор:

```
Klunk(int n = 0) { klunk_ct = n; }
```

Однако в классе может быть только один конструктор по умолчанию. То есть нельзя делать следующее:

```
Klunk() { klunk_ct = 0 }            // конструктор #1
Klunk(int n = 0) { klunk_ct = n; }  // неоднозначный конструктор #2
```

Почему это неоднозначно? Рассмотрим такие два объявления:

```
Klunk kar(10);    // в точности соответствует Klunt(int n)
Klunk bus;        // может соответствовать любому конструктору
```

Второе объявление соответствует как конструктору #1 (без аргументов), так и конструктору #2 (с аргументом по умолчанию, равным 0). Поэтому компилятор выдает сообщение об ошибке.

Конструкторы копирования

Конструктор копирования служит для копирования некоторого объекта в создаваемый объект. Другими словами, он используется во время инициализации — в том числе при передаче функции аргументов по значению — но не во время обычного присваивания. Конструктор копирования для класса обычно имеет следующий прототип:

```
Имя_класса(const Имя_класса &);
```

Обратите внимание, что в качестве аргумента он принимает константную ссылку на объект класса. Например, конструктор копирования для класса `String` будет выглядеть так:

```
StringBad(const StringBad &);
```

О конструкторе копирования нужно знать два момента: когда он используется и что он делает.

Когда используется конструктор копирования

Конструктор копирования вызывается всякий раз, когда создается новый объект, и для его инициализации берется значение существующего объекта того же типа. Это происходит в нескольких ситуациях. Наиболее очевидный случай — когда новый объект явно инициализируется существующим объектом. Например, если `motto` является объектом `StringBad`, то следующие четыре объявления вызывают конструктор копирования:

```
StringBad ditto(motto);            // вызывает StringBad(const StringBad &)
StringBad metoo = motto;           // вызывает StringBad(const StringBad &)
StringBad also = StringBad(motto); // вызывает StringBad(const StringBad &)
```

```
StringBad * pStringBad = new StringBad(motto);
     // вызывает StringBad(const StringBad &)
```

В зависимости от реализации, два объявления в середине могут использовать конструктор копирования либо непосредственно для создания объектов `metoo` и `also`, либо для генерирования временных объектов, содержимое которых затем присваивается объектам `metoo` и `also`. Приведенный выше код инициализирует анонимный объект значением `motto` и присваивает адрес нового объекта указателю `pstring`.

Менее очевидно то, что компилятор использует конструктор копирования при каждом генерировании копии объекта в программе. В частности, он применяется, когда функция передает объект по значению (как это делает функция `callme2()` в листинге 12.3) или когда функция возвращает объект. Ведь передача по значению подразумевает создание копии исходной переменной. Компилятор также использует конструктор копирования при генерировании временных объектов. Например, компилятор может генерировать временный объект `Vector` для хранения промежуточного результата при сложении трех объектов `Vector`. Различные компиляторы могут вести себя по-разному при создании временных объектов, но все они вызывают конструктор копирования при передаче объектов по значению и при их возврате. В частности, следующий вызов функции в листинге 12.3 запускает и конструктор копирования:

```
callme2(headline2);
```

В программе конструктор копирования применяется для инициализации `sb` — формального параметра типа `StringBad` для функции `callme2()`.

Кстати, тот факт, что при передаче объекта по значению вызывается конструктор копирования, является хорошей причиной для передачи по ссылке. Это позволит сэкономить время вызова конструктора и память для хранения нового объекта.

Что делает конструктор копирования по умолчанию

Конструктор копирования по умолчанию выполняет *почленное копирование* нестатических членов, также иногда называемое *поверхностным копированием*. Каждый член копируется по значению. Например, в листинге 12.3 оператор

```
StringBad sailor = sports;
```

эквивалентен следующему коду (который, правда, не скомпилируется по причине запрета доступа к закрытым членам):

```
StringBad sailor;
sailor.str = sports.str;
sailor.len = sports.len;
```

Если член сам является объектом класса, для копирования одного объекта-члена в другой используется конструктор копирования этого класса. Но это не влияет на статические члены, подобные `num_strings`, поскольку они принадлежат классу вообще, а не отдельным объектам. Действие неявного конструктора копирования показано на рис. 12.2.

Вернемся к `StringBad`: где в конструкторе копирования присутствует ошибка?

Теперь вы готовы понять две неточности в листинге 12.3. (Предположим, что вывод выглядит так, как показано сразу после листинга.) Первая неточность: согласно выводу программы, в ней удалено на два объекта больше, чем создано. Объясняется это тем, что программа создает два дополнительных объекта, используя конструктор копирования по умолчанию.

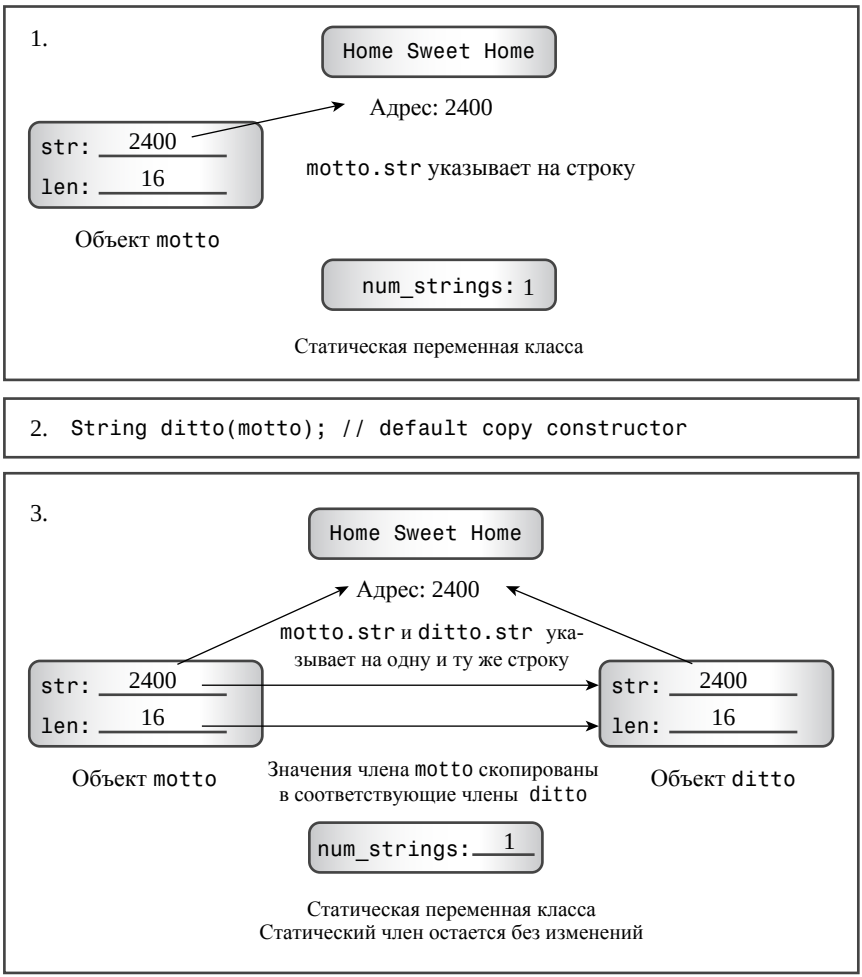


***Рис. 12.2.** Механизм почленного копирования*

Конструктор копирования применяется для инициализации формального параметра функции `callme2()` во время ее вызова, а также для инициализации объекта `sailor` объектом `sports`. Конструктор копирования по умолчанию никак не проявляет себя: не объявляет о создании объектов и не увеличивает счетчик `num_strings`. Однако деструктор обновляет счетчик и вызывается вплоть до уничтожения всех объектов, независимо от способа их создания. Отсюда и проблема – программа не может вести точный подсчет объектов. Для решения этой проблемы необходим явный конструктор копирования, который обновляет счетчик:

```
String::String(const String & s)
{
    num_strings++;
    ... // существенный код
}
```

Совет

Если в классе имеется статические данные-члены, значение которых изменяется при создании новых объектов, должен быть предусмотрен явный конструктор копирования, который принимает это внимание.

Вторая неточность более тонкая и опасная. Один из ее симптомов — бессмысленное содержимое строки:

```
headline2: Dû¤
```

Причина в том, что неявный конструктор копирования осуществляет копирование по значению. Рассмотрим, к примеру, листинг 12.3. В результате его работы выполняется следующий оператор:

```
sailor.str = sport.str;
```

Этот оператор копирует не строку, а указатель на строку. То есть после того как объекту `sailor` присвоено первоначальное значение `sports`, появилось два указателя на одну и ту же строку. Это не проблема, когда функция `operator<<()` использует указатель для вывода строки. Но это *становится* проблемой, когда вызывается деструктор. Ведь деструктор `StringBad` освобождает память, на которую указывает указатель `str`. Результат уничтожения `sailor`:

```
delete [] sailor.str;     // удаляется строка, на которую указывает ditto.str
```

Указатель `sailor.str` указывает на строку `"Spinach Leaves Bowl for Dollars"`, поскольку ему присвоено значение `sports.str`, которое указывает на данную строку. Поэтому операция `delete` освобождает память, занимаемую строкой `"Spinach Leaves Bowl for Dollars"`.

А после этого уничтожается объект `sports`:

```
delete [] sports.str;     // результат не определен
```

Здесь `sports.str` указывает на то поле памяти, которое уже очищено деструктором для объекта `sailor` — и поведение программы становится неопределенным или даже разрушительным. В случае листинга 12.3 программа выводит запорченные строки, что обычно является признаком неправильного управления памятью.

Еще одним неприятным симптомом является то, что попытка повторного удаления содержимого одного участка памяти может привести к аварийному завершению программы. Например, Microsoft Visual C++ 2010 (в отладочном режиме) выводит окно с сообщением об ошибке "Debug Assertion Failed!" (Отладочное утверждение не подтвердилось). А g++ 4.4.1 в Linux сообщает "double free or corruption" (повторное освобождение или запорчены данные) и прекращает работу. В других системах могут появляться другие сообщения или даже никакого сообщения вообще, но в таких программах содержится один и тот же дефект.

Устранение проблемы с помощью явного конструктора копирования

Для устранения проблем в структуре класса следует выполнять *глубокое копирование*. Это означает, что вместо простого копирования адреса строки конструктор копирования должен создать дубликат строки и присвоить адрес этого дубликата члену `str`. Тогда каждый объект получает собственную строку вместо ссылки на строку другого объекта, при каждом вызове деструктора освобождаются различные строки, и не происходит попыток повторного освобождения одной и той же строки. Вот как может выглядеть конструктор копирования `StringBad`:

```
StringBad::StringBad(const StringBad & st)
{
   num_strings++;                      // обновление статического члена
   len = st.len;                       // та же самая длина
   str = new char [len + 1];           // выделение памяти
   std:::strcpy(str, st.str);          // копирование строки в новое место
   cout << num_strings << ": \"" << str
        << "\" object created\n";      // для целей отладки
}
```

Определение конструктора копирования необходимо из-за того, что некоторые члены класса являются указателями на данные, инициализированными операцией new, а не самими данными. Глубокое копирование проиллюстрировано на рис. 12.3.

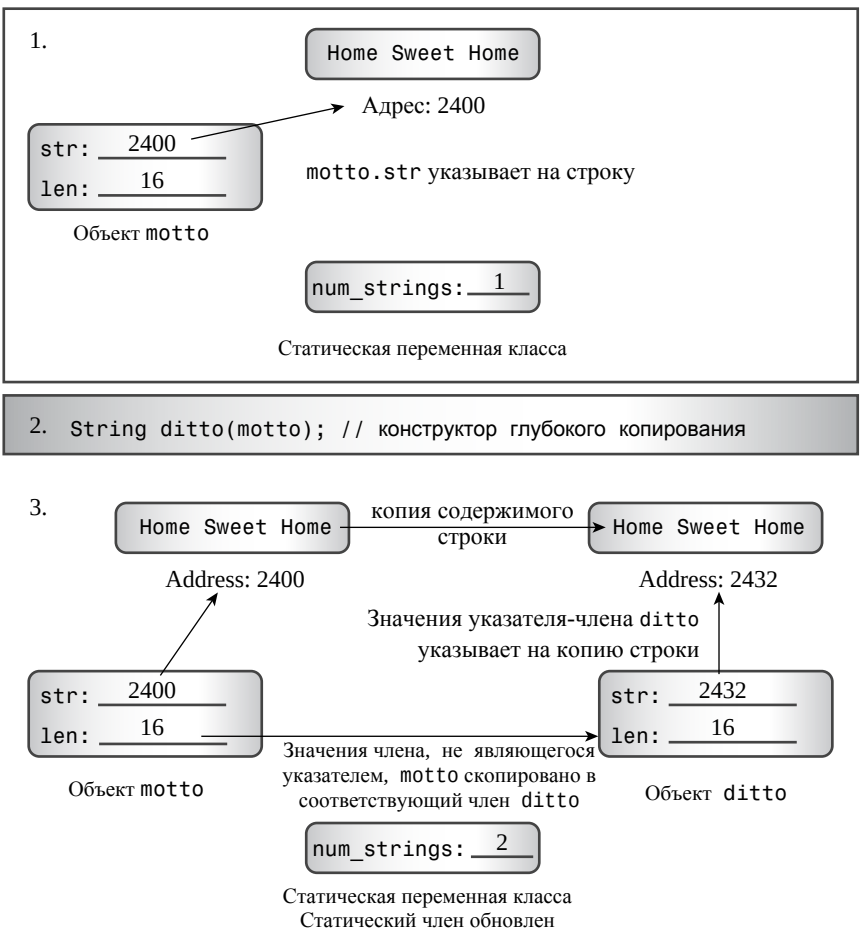


***Рис. 12.3.** Механизм глубокого копирования*

Внимание!

Если класс содержит члены, которые являются указателями, инициализированными операцией `new`, потребуется определить конструктор копирования, копирующий данные, на которые указывают указатели, а не сами указатели. Это называется *глубоким копированием*. Альтернативная форма копирования (*почленное* или *поверхностное копирование*) просто копирует значения указателей. Поверхностная копия — это только "наружное соскабливание" информации указателя для копирования, а не "глубокая добыча", требующая копирования конструкций, на которые указывают указатели.

Еще проблемы с `StringBad`: операции присваивания

Не все проблемы в листинге 12.3 можно списать на конструктор копирования по умолчанию; следует обратить внимание и на операцию присваивания по умолчанию. Подобно тому, как ANSI C разрешает присваивание структур, C++ допускает присваивание объектов класса. Это делается за счет автоматической перегрузки операции присваивания для класса, которая имеет следующий прототип:

```
Имя_класса & Имя_класса::operator=(const Имя_класса &);
```

Другими словами, она принимает и возвращает ссылку на объект класса. Например, прототип для класса `StringBad` выглядит так:

```
StringBad & StringBad::operator=(const StringBad &);
```

Когда используется операция присваивания и что она делает

Перегруженная операция присваивания используется при присваивании одного объекта другому существующему объекту:

```
StringBad headline1("Celery Stalks at Midnight");
...
StringBad knot;
knot = headline1;         // вызывается операция присваивания
```

При инициализации объекта операция присваивания не обязательна:

```
StringBad metoo = knot;   // используется конструктор копирования,
                          // но возможно и присваивание
```

Здесь `metoo` — только что созданный объект, который инициализирован значениями из `knot`; следовательно, используется конструктор копирования. Однако, как уже было сказано, реализации могут выполнять такую операцию в два этапа: создание временного объекта с помощью конструктора копирования и затем копирование значений в новый объект с помощью присваивания. То есть инициализация всегда вызывает конструктор копирования, а формы, использующие операцию =, могут также вызывать операцию присваивания.

Как и в случае конструктора копирования, неявная реализация операции присваивания выполняет почленное копирование. Если какой-то член сам является объектом некоторого класса, то программа использует операцию присваивания, определенную для данного класса, чтобы выполнить копирование для данного конкретного члена. На статические члены данных это не распространяется.

Где присваивание в `StringBad` работает неправильно

В листинге 12.3 объекту `knot` присваивается значение `headline1`:

```
knot = headline1;
```

Когда для `knot` вызывается деструктор, он выводит следующее сообщение:

```
"Celery Stalks at Midnight" object deleted, 2 left
```

Когда деструктор вызывается для headline1, он выводит:

```
"-|" object deleted, -2 left
```

(Многие реализации аварийно завершают работу еще до этого.)

Здесь присутствует та же проблема, что и с неявным конструктором копирования — запорченные данные. И снова все упирается в почленное копирование, когда и headline1.str, и knot.str указывают на один и тот же адрес. При вызове деструктора для knot строка "Celery Stalks at Midnight" удаляется, а при вызове деструктора для headline1 программа пытается удалить уже удаленную строку. Как отмечалось ранее, результат попытки удаления ранее удаленных данных не определен: это может изменить содержимое памяти либо привести к аварийному завершению программы. А если результат отдельной операции не определен, то компилятор может делать все, что ему заблагорассудится, включая вывод на экран свежих анекдотов или удаление с жесткого диска нефотогеничных файлов. Очевидно, что компиляторы не предназначены для подобного рода действий.

Исправление присваивания

Для решения проблем, возникающих из-за некорректных стандартных операций присваивания, можно определить собственную операцию присваивания, которая выполняет глубокое копирование. Реализация аналогична конструктору копирования, кроме нескольких отличий, которые описаны ниже.

- Поскольку целевой объект может ссылаться на данные, для которых уже была распределена память, функция должна использовать операцию delete [] для ее освобождения.
- Функция должна содержать защиту от присваивания объекта самому себе — иначе вышеописанное освобождение памяти может стереть содержимое объекта до того, как оно будет переустановлено.
- Функция возвращает ссылку на вызывающий объект.

Возвращая объект, функция может эмулировать цепочку обычных присваиваний для встроенных типов. То есть если S0, S1 и S2 являются объектами StringBad, то можно записать

```
S0 = S1 = S2;
```

В нотации с помощью функций это выглядит так:

```
S0.operator=(S1.operator=(S2));
```

Таким образом, значение, возвращаемое функцией S1.operator=(S2), становится аргументом функции S0.operator=(). Поскольку возвращаемое значение является ссылкой на объект String, это корректный тип аргумента.

А вот как можно реализовать операцию присваивания для класса StringBad:

```
StringBad & StringBad::operator=(const StringBad & st)
{
   if (this == &st)              // присваивание объекта самому себе
      return *this;              // все готово
   delete [] str;                // освобождение старой строки
   len = st.len;
   str = new char [len + 1];     // выделение памяти для новой строки
   std::strcpy(str, st.str);     // копирование строки
   return *this;                 // возврат ссылки на вызывающий объект
}
```

Сначала код проверяет, не выполняется ли присваивание самому себе. Для этого адрес в правой части присваивания (`&s`) сравнивается с адресом принимающего объекта (`this`). Если они совпадают, функция возвращает `*this` и завершает работу. Как вы помните, в главе 10 было сказано, что операция присваивания может быть перегружена только с помощью функции-члена класса.

Иначе функция переходит к освобождению памяти, на которую указывает `str`. Ведь после этого указателю `str` будет присвоен адрес новой строки. Если не выполнить сначала операцию `delete`, то предыдущая строка останется в памяти, а поскольку указатель на старую строку уже не существует, память будет занята зря.

Далее функция действует как конструктор копирования: выделяет достаточный объем памяти для новой строки и копирует строку из объекта в правой части в новое место.

После этого функция возвращает `*this` и завершается.

Присваивание не создает новый объект, поэтому корректировать значение статического члена данных `num_strings` не нужно.

Добавление в класс `StringBad` операции присваивания и описанного выше конструктора копирования устраняет все проблемы. Ниже показано несколько последних строк выходных данных, которые получены после всех указанных изменений:

```
End of main()
"Celery Stalks at Midnight" object deleted, 4 left
"Spinach Leaves Bowl for Dollars" object deleted, 3 left
"Spinach Leaves Bowl for Dollars" object deleted, 2 left
"Lettuce Prey" object deleted, 1 left
"Celery Stalks at Midnight" object deleted, 0 left
```

Теперь подсчет объектов ведется правильно, и ни одна из строк не искажается.

Новый усовершенствованный класс `String`

Теперь наших знаний уже хватит на модифицированный класс `StringBad`, который мы назовем просто `String`. Во-первых, нужно добавить рассмотренные выше конструктор копирования и операцию присваивания, чтобы класс корректно управлял памятью, используемой объектами класса. Во-вторых, мы уже знаем, когда создаются и уничтожаются объекты, и можно "лишить права голоса" конструкторы и деструкторы, чтобы они больше не сообщали о своем использовании. Кроме того, можно упростить конструктор по умолчанию, чтобы он создавал не строку `"C++"`, а пустую строку.

После этого в класс можно добавить некоторые новые средства. Полезный класс `String` может содержать все функциональные возможности стандартной библиотеки `cstring` строковых функций, но мы добавим только такие, которые помогут увидеть механизм работы. (Ведь класс `String` — учебный пример, а стандартный класс `string` из C++ значительно шире.) А именно, мы добавим следующие методы:

```
int length () const { return len; }
friend bool operator<(const String &st, const String &st2);
friend bool operator>(const String &st1, const String &st2);
friend bool operator==(const String &st, const String &st2);
friend operator>>(istream & is, String & st);
char & operator[](int i);
const char & operator[](int i) const;
static int HowMany();
```

Первый из новых методов возвращает длину хранимой строки. Следующие три дружественных функции позволяют сравнивать строки. Функция `operator>>()` обеспечивает возможности простого ввода. Две функции `operator[]()` предоставляют доступ к отдельным символам строки в виде массива. Статический метод класса `HowMany()` дополняет статический член данных класса `num_strings`. А теперь рассмотрим каждый из них подробнее.

Пересмотренный конструктор по умолчанию

Новый конструктор по умолчанию выглядит следующим образом:

```
String::String()
{
   len = 0;
   str = new char[1];
   str[0] = '\0';    // строка по умолчанию
}
```

Вас может заинтересовать, почему в коде применяется оператор

```
str = new char[1];
```

а не

```
str = new char;
```

Обе формы выделяют одинаковый объем памяти. Различие состоит в том, что первая форма совместима с деструктором класса, а вторая нет. Вспомните, что деструктор содержит следующий код:

```
delete [] str;
```

Использование операции `delete []` совместимо с указателями, инициализированными операцией `new []`, и с нулевым указателем. Поэтому еще одним вариантом является замена кода

```
str = new char[1];
str[0] = '\0';       // строка по умолчанию
```

кодом

```
str = 0;             // теперь str — нулевой указатель
```

Результат использования `delete []` с любыми указателями, инициализированными любым другим способом, не определен:

```
char words[15] = "bad idea";
char * p1 = words;
char * p2 = new char;
char * p3;
delete [] p1;        // не определено, поэтому не делайте так
delete [] p2;        // не определено, поэтому не делайте так
delete [] p3;        // не определено, поэтому не делайте так
```

Нулевой указатель в C++11

В C++98 литерал `0` имеет две трактовки: числовое значение `0` и нулевой указатель. Это усложняет понимание кода и читателям, и компиляторам. Иногда программисты употребляют конструкцию `(void *) 0`, чтобы подчеркнуть, что это именно указатель. (Сам нулевой указатель может иметь и ненулевое внутреннее представление.) Другие программисты используют макрос `NULL`, определенный в языке C для представления нулевого указателя.

Но эти решения все-таки неполны. В C++11 введено лучшее решение — ключевое слово `nullptr`, которое означает нулевой указатель. Вы можете, как и раньше, записывать просто `0` — иначе придется пересмотреть огромные объемы существующего кода — но с данного момента рекомендуется использовать `nullptr`:

```
str = nullptr;  // нотация нулевого указателя в C++11
```

Члены для сравнений

Три метода в классе `String` выполняют сравнения. Функция `operator<()` возвращает значение `true`, если первая строка идет раньше второй в алфавитном порядке (точнее, в машинной последовательности сопоставления). Наиболее простой способ реализации функций сравнения строк – использование стандартной функции `strcmp()`. Она возвращает отрицательное значение, если первый аргумент предшествует второму по алфавиту, `0`, если строки одинаковые, и положительное значение, если первая строка по алфавиту следует за второй. Функцию `strcmp()` можно задействовать следующим образом:

```
bool operator<(const String &st1, const String &st2)
{
   if (std::strcmp(st1.str, st2.str) > 0)
      return true;
   else
      return false;
}
```

Поскольку встроенная операция > уже возвращает значение типа `bool`, можно дополнительно упростить код:

```
bool operator<(const String &st1, const String &st2)
{
   return (std::strcmp(st1.str, st2.str) < 0);
}
```

По аналогии можно записать и две остальные функции сравнения:

```
bool operator>(const String &st1, const String &st2)
{
   return st2.str < st1.str;
}
bool operator==(const String &st1, const String &st2)
{
   return (std::strcmp(st1.str, st2.str) == 0);
}
```

Первое определение выражает операцию > через операцию < и может служить хорошим кандидатом на встроенную функцию.

Создание дружественных функций сравнения облегчает сравнение объектов `String` и стандартных строк C. Пусть, например, `answer` – объект `String`, и имеется следующий код:

```
if ("love" == answer)
```

Он транслируется в такой код:

```
if (operator==("love", answer))
```

Затем компилятор использует один из конструкторов для преобразования кода к следующему виду:

```
if (operator==(String("love"), answer))
```

И это как раз соответствует прототипу.

Доступ к символам с помощью скобочной нотации

В стандартных строках стиля C можно обращаться к отдельным символам с помощью квадратных скобок:

```
char city[40] = "Amsterdam";
cout << city[0] << endl;     // отображает букву A
```

В C++ две квадратные скобки образуют одну операцию — скобочную, которую можно перегрузить с помощью метода `operator[]()`. Как правило, бинарная операция C++ (с двумя операндами) предусматривает наличие знака операции между двумя операндами, например, `2 + 5`. А в случае скобочной операции один операнд располагается перед первой скобкой, а второй — между двумя скобками. Например, в выражении `city[0]` первый операнд — это `city`, `[]` — операция, а `0` — второй операнд.

Пусть `opera` является объектом `String`:

```
String opera("The Magic Flute");
```

Если в коде имеется выражение `opera[4]`, C++ ищет метод со следующим именем и сигнатурой:

```
operator[](int i)
```

Если такой прототип найден, компилятор заменяет выражение `opera[4]` вызовом данной функции:

```
opera.operator[](4)
```

Объект `opera` вызывает метод, а индекс массива `4` становится аргументом функции.

Вот пример простой реализации скобочной операции:

```
char & String::operator[](int i)
{
   return str[i];
}
```

При таком определении оператор

```
cout << opera[4];
```

транслируется в

```
cout << opera.operator[](4);
```

При этом возвращается значение `opera.str[4]`, т.е. символ `'e'`. Подобным образом открытый метод предоставляет доступ к закрытым данным.

Если объявить возвращаемый тип как `char &`, то это позволит присваивать значения отдельным элементам. Например, можно использовать следующий код:

```
String means("might");
means[0] = 'r';
```

Второй оператор преобразуется в вызов функции перегруженной операции:

```
means.operator[][0] = 'r';
```

Это код присваивает `'r'` возвращаемому значению метода. Но функция возвращает ссылку на `means.str[0]`, поэтому данный код эквивалентен следующему:

```
means.str[0] = 'r';
```

Последняя строка кода нарушает закрытый доступ, но операция operator[]() является методом класса, и она допускает изменения содержимого массива. В результате строка "might" становится "right".

Предположим, что имеется константный объект:

```
const String answer("futile");
```

Тогда если единственным доступным определением operator[]() является приведенное выше, то следующий код будет помечен как ошибочный:

```
cout << answer[1];  // ошибка компиляции
```

Дело в том, что объект answer объявлен как константный, а метод не обещает не менять данные. (Ведь зачастую методы как раз и создаются для изменения данных — потому он и не обещает.)

Однако при перегрузке C++ может различить сигнатуры константных и не константных функций, поэтому можно предусмотреть вторую версию operator[](), которая будет использоваться только объектами const String:

```
// Для использования с объектами const String
const char & String::operator[](int i) const
{
   return str[i];
}
```

Такие определения позволят иметь доступ для чтения и записи к обычным объектам String и доступ только для чтения к данным const String:

```
String text("Once upon a time");
const String answer("futile");
cout << text[1];  // нормально, используется не константная версия operator[]()
cout << answer[1]; // нормально, используется константная версия operator[]()
cin >> text[1];   // нормально, используется не константная версия operator[]()
cin >> answer[1]; // ошибка компиляции
```

Статические функции-члены класса

Функцию-член можно объявить как статическую. (Ключевое слово static должно присутствовать в объявлении, а не в определении функции, если последнее размещается отдельно.) Это влечет за собой два важных следствия.

Во-первых, статическую функцию-член не обязательно вызывать через объект, она даже не получает указатель this. Если статическая функция-член объявляется в разделе public, то ее можно вызвать с помощью имени класса и операции разрешения контекста. К примеру, в класс String можно добавить статическую функцию-член с именем HowMany() и следующим прототипом/определением в объявлении класса:

```
static int HowMany() { return num_strings; }
```

Вызвать ее можно так:

```
int count = String::HowMany(); // вызов статической функции-члена
```

Во-вторых, поскольку статическая функция-член не связана с каким-либо конкретным объектом, то она может использовать только статические члены данных. Например, статический метод HowMany() может получить доступ к статическому члену num_strings, но не может — к членам str или len.

Аналогично статическая функция-член может применяться для установки флага, глобального для класса, который управляет поведением каких-то аспектов интерфейса класса. Например, он может управлять форматированием, которое использует метод, отображающий содержимое класса.

Дополнительная перегрузка операции присваивания

Прежде чем рассматривать новые листинги с примерами для класса `String`, обсудим еще один вопрос. Предположим, что необходимо скопировать обычную строку в объект `String`. Например, строка читается с помощью функции `getline()`, а потом помещается в объект `String`. Методы класса уже позволяют сделать следующее:

```
String name;
char temp[40];
cin.getline(temp, 40);
name = temp;  // преобразование типа с помощью конструктора
```

Однако если делать это часто, такое решение может оказаться неудовлетворительным. Чтобы понять почему, давайте посмотрим, как работают эти операторы.

1. Программа использует конструктор `String(const char *)` для создания временного объекта `String`, содержащего копию строки, которая хранится в `temp`. Вспомните из главы 11, что конструктор с одним аргументом работает как функция преобразования.
2. В листинге 12.6 (ниже в данной главе) программа использует функцию `String & String::operator=(const String &)` для копирования информации из временного объекта в объект `name`.
3. Программа вызывает деструктор `~String()` для удаления временного объекта.

Самым простым способом увеличения эффективности процесса является перегрузка операции присваивания таким образом, чтобы она работала непосредственно с обычными строками. Это устранит дополнительные шаги по созданию и удалению временного объекта. Ниже показана одна из возможных реализаций:

```
String & String::operator=(const char * s)
{
   delete [] str;
   len = std::strlen(s);
   str = new char[len + 1];
   std::strcpy(str, s);
   return *this;
}
```

Как обычно, необходимо освободить память, ранее управляемую указателем `str`, и выделить достаточный объем памяти для новой строки.

В листинге 12.4 показано пересмотренное объявление класса. В дополнение к уже упомянутым изменениям, в нем определяется константа `CINLIM`, которая используется в реализации `operator>>()`.

Листинг 12.4. `string1.h`

```
// string1.h -- исправленное и расширенное объявление строкового класса
#ifndef STRING1_H_
#define STRING1_H_
#include <iostream>
using std::ostream;
using std::istream;

class String
{
private:
   char * str;                 // указатель на строку
```

```
    int len;                          // длина строки
    static int num_strings;           // количество объектов
    static const int CINLIM = 80;     // предел ввода для cin
public:
    // Конструкторы и другие методы
    String(const char * s);           // конструктор
    String();                         // конструктор по умолчанию
    String(const String &);           // конструктор копирования
    ~String();                        // деструктор
    int length () const { return len; }

    // Методы перегруженных операций
    String & operator=(const String &);
    String & operator=(const char *);
    char & operator[](int i);
    const char & operator[](int i) const;

    // Дружественные функции перегруженных операций
    friend bool operator<(const String &st, const String &st2);
    friend bool operator>(const String &st1, const String &st2);
    friend bool operator==(const String &st, const String &st2);
    friend ostream & operator<<(ostream & os, const String & st);
    friend istream & operator>>(istream & is, String & st);

    // Статическая функция
    static int HowMany();
};
#endif
```

В листинге 12.5 представлены пересмотренные определения методов.

Листинг 12.5. `string1.cpp`

```
// string1.cpp -- методы класса String
#include <cstring>          // в некоторых случаях — string.h
#include "string1.h"        // включение <iostream>
using std::cin;
using std::cout;

// Инициализация статического члена класса
int String::num_strings = 0;

// Статический метод
int String::HowMany()
{
    return num_strings;
}

// Методы класса
String::String(const char * s)          // создание String из C-строки
{
    len = std::strlen(s);               // установка размера
    str = new char[len + 1];            // выделение памяти
    std::strcpy(str, s);                // инициализация указателя
    num_strings++;                      // корректировка счетчика объектов
}

String::String()                        // конструктор по умолчанию
{
    len = 4;
    str = new char[1];
    str[0] = '\0';                      // строка по умолчанию
    num_strings++;
}
```

```
String::String(const String & st)
{
   num_strings++;                  // обработка обновления статического члена
   len = st.len;                   // длина та же
   str = new char [len + 1];       // выделение памяти
   std::strcpy(str, st.str);       // копирование строки в новое место
}
String::~String()                  // необходимый деструктор
{
   --num_strings;                  // требуется
   delete [] str;                  // требуется
}
// Методы перегруженных операций
// Присваивание объекта String объекту String
String & String::operator=(const String & st)
{
   if (this == &st)
      return *this;
   delete [] str;
   len = st.len;
   str = new char[len + 1];
   std::strcpy(str, st.str);
   return *this;
}
// Присваивание C-строки объекту String
String & String::operator=(const char * s)
{
   delete [] str;
   len = std::strlen(s);
   str = new char[len + 1];
   std::strcpy(str, s);
   return *this;
}
// Доступ для чтения и записи отдельных символов в неконстантном объекте String
char & String::operator[](int i)
{
   return str[i];
}
// Доступ только для чтения отдельных символов в константном объекте String
const char & String::operator[](int i) const
{
   return str[i];
}
// Дружественные функции перегруженных операций
bool operator<(const String &st1, const String &st2)
{
   return (std::strcmp(st1.str, st2.str) < 0);
}
bool operator>(const String &st1, const String &st2)
{
   return st2.str < st1.str;
}
bool operator==(const String &st1, const String &st2)
{
   return (std::strcmp(st1.str, st2.str) == 0);
}
```

```
// Простой вывод String
ostream & operator<<(ostream & os, const String & st)
{
   os << st.str;
   return os;
}
// Простой ввод String
istream & operator>>(istream & is, String & st)
{
   char temp[String::CINLIM];
   is.get(temp, String::CINLIM);
   if (is)
     st = temp;
   while (is && is.get() != '\n')
     continue;
   return is;
}
```

Перегруженная операция >> обеспечивает простой способ ввода строки с клавиатуры в объект `String`. Она принимает введенную строку длиной `String::CINLIM` или менее символов и отбрасывает все символы сверх этого предела. Учтите, что значение объекта `istream` в условии `if` равно `false`, если ввод данных по каким-то причинам аварийно прерывается — например, появление условия конца файла или, в случае `get(char *, int)`, чтение пустой строки.

Короткая программа, приведенная в листинге 12.6, проверяет класс `String`, позволяя ввести несколько строк. Программа запрашивает у пользователя ввод поговорок, помещает строки в объекты `String`, выводит их и выдает отчет о том, какая строка самая короткая, и какая идет первой в алфавитном порядке.

Листинг 12.6. `sayings1.cpp`

```
// sayings1.cpp -- использование расширенного класса String
// компилировать вместе с string1.cpp
#include <iostream>
#include "string1.h"
const int ArSize = 10;
const int MaxLen =81;
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   String name;
   cout <<"Hi, what's your name?\n>> ";                  // ввод имени
   cin >> name;
   cout << name << ", please enter up to " << ArSize
        << " short sayings <empty line to quit>:\n";    // ввод поговорки
   String sayings[ArSize];    // массив объектов
   char temp[MaxLen];         // временное хранилище для строки
   int i;
   for (i = 0; i < ArSize; i++)
   {
      cout << i+1 << ": ";
      cin.get(temp, MaxLen);
      while (cin && cin.get() != '\n')
         continue;
```

```
      if (!cin || temp[0] == '\0')           // пустая строка?
         break;                              // i не инкрементируется
      else
         sayings[i] = temp;                  // перегруженное присваивание
   }
   int total = i; // общее количество прочитанных строк
   if ( total > 0)
   {
      cout << "Here are your sayings:\n";    // вывод поговорок
      for (i = 0; i < total; i++)
         cout << sayings[i][0] << ": " << sayings[i] << endl;
      int shortest = 0;
      int first = 0;
      for (i = 1; i < total; i++)
      {
         if (sayings[i].length() < sayings[shortest].length())
            shortest = i;
         if (sayings[i] < sayings[first])
            first = i;
      }
      cout << "Shortest saying:\n" << sayings[shortest] << endl;
           // Самая короткая поговорка
      cout << "First alphabetically:\n" << sayings[first] << endl;
           // Первая по алфавиту
      cout << "This program used "<< String::HowMany()
           << " String objects. Bye.\n";
             // Количество используемых объектов String
   }
   else
      cout << "No input! Bye.\n";            // ничего не было введено
   return 0;
}
```

На заметку!

Более старые версии `get(char *, int)` не устанавливают значение `false` при чтении пустой строки. В таких версиях при вводе пустой строки первым символом в строке считается нулевой символ. В данном примере используется следующий код:

```
if (!cin || temp[0] == '\0')     // пустая строка?
   break;                        // i не инкрементируется
```

Если реализация поддерживает текущий стандарт C++, то пустая строка обнаруживается при первой проверке в операторе `if`, а в более старых реализациях она обнаруживается при второй проверке.

Программа из листинга 12.6 предлагает пользователю ввести до 10 поговорок. Каждая поговорка считывается во временный символьный массив, а затем копируется в объект `String`. Если пользователь вводит пустую строку, оператор `break` завершает цикл ввода. После вывода введенных данных программа использует функции-члены `length()` и `operator<()` для нахождения самой короткой и самой первой в алфавитном порядке строки. Программа также применяет операцию индексации (`[]`) для того, чтобы разместить перед каждой поговоркой ее начальный символ. Рассмотрим пример выполнения этой программы:

```
Hi, what's your name?
>> Misty Gutz
```

```
Misty Gutz, please enter up to 10 short sayings <empty line to quit>:
1: a fool and his money are soon parted
2: penny wise, pound foolish
3: the love of money is the root of much evil
4: out of sight, out of mind
5: absence makes the heart grow fonder
6: absinthe makes the hart grow fonder
7:
Here are your sayings:
a: a fool and his money are soon parted
p: penny wise, pound foolish
t: the love of money is the root of much evil
o: out of sight, out of mind
a: absence makes the heart grow fonder
a: absinthe makes the hart grow fonder
Shortest saying:
penny wise, pound foolish
First alphabetically:
a fool and his money are soon parted
This program used 11 String objects. Bye.
```

О чем следует помнить при использовании операции new в конструкторах

Теперь вы уже понимаете, что использование операции new для инициализации указателей-членов объекта требует особой внимательности. В частности, вы должны следовать таким рекомендациям.

- Если для инициализации указателя-члена в конструкторе применяется операция new, то в деструкторе нужно использовать операцию delete.
- Операции new и delete должны быть согласованными. Операции new должна соответствовать операция delete, а операции new [] — операция delete [].
- Если применяется несколько конструкторов, все они должны единообразно использовать операцию new — либо все со скобками, либо все без скобок. В классе существует только один деструктор, и все конструкторы должны быть совместимы с ним. При этом допустимо инициализировать указатель с помощью операции new в одном конструкторе и с помощью нулевого указателя (NULL или nullptr в C++11) — в другом, поскольку к нулевому указателю можно применять операцию delete (со скобками или без них).

NULL, 0 или nullptr?

Исторически сложилось так, что нулевой указатель может быть представлен как 0 или как NULL (символическая константа, определенная как 0 во многих заголовочных файлах). Программисты, пишущие на C, часто используют NULL вместо 0 в качестве визуального напоминания о том, что значение является указателем, подобно тому, как '\0' применяется вместо 0 для обозначения нулевого символа — визуальное напоминание о том, что значение является символом. Однако в C++ традиционно отдают предпочтение простому 0 вместо эквивалентного ему NULL. И, как уже упоминалось, в C++11 имеется лучший вариант — ключевое слово nullptr.

- Необходимо определить конструктор копирования, в котором инициализация одного объекта другим выполняется с помощью глубокого копирования. Обычно конструктор должен быть построен по следующему образцу:

```
String::String(const String & st)
{
   num_strings++;              // при необходимости обработка обновления
                               // статического члена
   len = st.len;               // та же длина, что и у копируемой строки
   str = new char [len + 1];   // выделение памяти
   std::strcpy(str, st.str);  // копирование строки в новое место
}
```

То есть конструктор копирования должен выделять память для хранения копируемых данных и копировать эти данные, а не только их адрес. Кроме того, он должен обновлять все статические члены класса, чьи значения затрагиваются данных процессом.

- Необходимо определить операцию присваивания, в которой копирование одного объекта в другой осуществляется с помощью глубокого копирования. Обычно метод класса должен быть построен по следующему образцу:

```
String & String::operator=(const String & st)
{
   if (this == &st)            // присваивание объекта самому себе
     return *this;             // готово
   delete [] str;              // освобождение старой строки
   len = st.len;
   str = new char [len + 1];   // получение памяти для новой строки
   std::strcpy(str, st.str);  // копирование строки
   return *this;               // возврат ссылки на вызвавший объект
}
```

То есть метод должен проверить наличие присваивания объекта самому себе, освободить память, на которую ранее указывал указатель-член, скопировать данные, а не только их адрес, и возвратить ссылку на вызвавший объект.

Что следует делать, а что делать нельзя

В следующем фрагменте кода представлены два примера, показывающие, чего делать не стоит, и один пример правильного конструктора:

```
String::String()
{
   str = "default string";     // неверно: не хватает new []
   len = std::strlen(str);
}

String::String(const char * s)
{
   len = std::strlen(s);
   str = new char;             // неверно: не хватает []
   std::strcpy(str, s);        // неверно: некуда же
}

String::String(const String & st)
{
   len = st.len;
   str = new char[len + 1];    // правильно: выделение памяти
   std::strcpy(str, st.str);  // правильно: копируется значение
}
```

В первом конструкторе не хватает вызова new для инициализации str. Деструктор, вызываемый для объекта, применяет к str операцию delete. Результат использования операции delete с указателем, который не был инициализирован с помощью new, не определен, но вряд ли он будет хорошим. Подойдет один из следующих вариантов:

```
String::String()
{
   len = 0;
   str = new char[1];              // используется new с []
   str[0] = '\0';
}
String::String()
{
   len = 0;
   str = 0;                        // или str = nullptr; в C++11
}
String::String()
{
   static const char * s = "C++";  // инициализируется только однажды
   len = std::strlen(s);
   str = new char[len + 1];        // использует new с []
   std::strcpy(str, s);
}
```

Второй конструктор в исходном фрагменте выполняет операцию new, но не запрашивает нужный объем памяти. Поэтому операция new возвращает блок памяти, способный вместить только один символ. При попытке скопировать в это место более длинную строку возникнут проблемы с памятью. К тому же использование new без скобок несовместимо с правильной формой других конструкторов.

Третий конструктор ошибок не содержит.

В завершение рассмотрим пример деструктора, который *не будет* правильно работать с приведенными ранее конструкторами:

```
String::~String()
{
   delete str;       // неверно, нужно использовать delete [] str;
}
```

В деструкторе неправильно используется delete. Поскольку конструкторы запрашивают массив символов, деструктор должен удалять массив.

Почленное копирование для классов с членами других классов

Предположим, что класс String или даже стандартный класс string используется в качестве типа для членов другого класса:

```
class Magazine
{
private:
   String title;
   string publisher;
...
};
```

И String, и string используют динамическое выделение памяти. Значит ли это, что для класса Magazine нужно писать специальный конструктор копирования и

операцию присваивания? Оказывается, нет – по крайней мере, не в нем самом. С такой задачей способно справиться стандартное поведение почленного копирования и присваивания. Когда осуществляется копирование или присваивание одного объекта `Magazine` другому, почленное копирование использует конструкторы копирования и операции присваивания, определенные для типов членов такого объекта. Это значит, что для копирования члена `title` из одного объекта в другой будет задействован конструктор копирования `String`, для присваивания одного объекта `Magazine` другому – операция присваивания и т.д. Правда, все несколько усложняется, если в классе `Magazine` требуется конструктор копирования и операция присваивания для некоторых других членов класса. Тогда эти функции должны явно вызывать конструкторы копирования и операции присваивания для классов `String` и `string`. Этот вопрос будет рассмотрен в главе 13.

Замечания о возвращаемых объектах

При возврате объекта функцией-членом или автономной функцией возможны следующие варианты. Функция может возвращать ссылку на объект, константную ссылку на объект, объект или константный объект. Вы уже видели все примеры, кроме последнего, поэтому сейчас самое время рассмотреть и его.

Возврат ссылки на константный объект

Основной причиной использования константной ссылки является производительность, но здесь имеется несколько ограничений. Если функция возвращает объект, который передан ей (либо путем вызова объекта, либо в качестве аргумента метода), то можно увеличить эффективность метода, возвращая из него ссылку. Например, предположим, что требуется написать функцию `Max()`, которая возвращает больший из двух объектов `Vector`, где `Vector` – это класс, разработанный в главе 11. Функция используется следующим образом:

```
Vector force1(50,60);
Vector force2(10,70);
Vector max;
max = Max(force1, force2);
```

Обе следующие реализации будут работать:

```
// версия 1
Vector Max(const Vector & v1, const Vector & v2)
{
   if (v1.magval() > v2.magval())
      return v1;
   else
      return v2;
}
// версия 2
const Vector & Max(const Vector & v1, const Vector & v2)
{
   if (v1.magval() > v2.magval())
      return v1;
   else
      return v2;
}
```

Здесь необходимо отметить три важных момента. Во-первых, вспомните, что при возврате объекта вызывается конструктор копирования, а при возврате ссылки – нет.

Поэтому второй вариант выполняет меньше работы и более эффективен. Во-вторых, при выполнении вызываемой функции ссылка должна указывать на существующий объект. В данном примере ссылка формируется либо на force1, либо на force2, причем оба объекта определены в вызывающей функции, поэтому указанное требование выполняется. В-третьих, v1 и v2 объявлены как константные ссылки — соответственно и возвращаемый тип должен быть константным.

Возврат ссылки на не константный объект

Два популярных примера возврата не константного объекта — перегрузка операции присваивания и перегрузка операции << для использования с cout. Первое делается по соображениям повышения производительности, а второе — при необходимости.

Значение, возвращаемое operator=(), используется для присваивания в виде цепочки:

```
String s1("Good stuff");
String s2, s3;
s3 = s2 = s1;
```

В данном коде значение, возвращаемое s2.operator=(s1), присваивается s3. При этом можно использовать как объект String, так и ссылку на объект String. Однако, как и в примере с объектом Vector, применение ссылки позволяет функции не вызывать конструктор копирования String для создания нового объекта String. В этом случае возвращаемый тип не является константным, поскольку метод operator=() возвращает ссылку на измененный объект s2.

Значение, возвращаемое operator<<(), также применяется для присваивания в виде цепочки:

```
String s1("Good stuff");
cout << s1 << "is coming!";
```

Здесь значение, возвращаемое методом operator<<(cout, s1), становится объектом, который используется для вывода строки "is coming!". Возвращаемый тип должен быть ostream &, а не просто ostream. Использование типа ostream потребует вызова конструктора копирования ostream, но оказывается, что класс ostream не имеет открытого конструктора копирования. К счастью, возврат ссылки на cout не вызывает проблем, поскольку cout уже содержится в области действия вызывающей функции.

Возврат объекта

Если возвращаемый объект является локальным для вызванной функции, он не должен возвращаться по ссылке, поскольку при завершении функции для него вызывается собственный деструктор. Таким образом, когда управление возвращается в вызвавшую функцию, объект, на который может указывать ссылка, уже не существует. В таком случае следует возвращать объект, а не ссылку. Как правило, в эту категорию попадают перегруженные арифметические операции. Рассмотрим пример, в котором снова используется класс Vector:

```
Vector force1(50,60);
Vector force2(10,70);
Vector net;
net = force1 + force2;
```

Возвращаемое значение не является ни force1, ни force2, которые должны остаться неизменными после обработки. Поэтому возвращаемое значение не может

быть ссылкой на объект, который уже существует в вызывающей функции. Сумма векторов — это новый временный объект, вычисляемый в `Vector::operator+()`, а функция не должна возвращать ссылку на временный объект. Она должна возвращать сам векторный объект, а не ссылку на него:

```
Vector Vector::operator+(const Vector & b) const
{
    return Vector(x + b.x, y + b.y);
}
```

Здесь возникают дополнительные затраты на вызов конструктора копирования для создания возвращаемого объекта, но это неизбежно.

И еще одно наблюдение: в примере `Vector::operator+()` вызов конструктора `Vector(x+ b.x, y + b.y)` создает объект, доступный методу `operator+()`. А неявный вызов конструктора копирования, порождаемый оператором `return`, создает объект, доступный вызывающей программе.

Возврат константного объекта

Предыдущее определение `Vector::operator+()` обладает странным свойством. Предполагается следующее использование операции:

```
net = force1 + force2;      // 1: три объекта Vector
```

Однако данное определение позволяет использовать и такие операторы:

```
force1 + force2 = net;      // 2: непроизносимое программирование
cout << (force1 + force2 = net).magval() << endl;
                            // 3: невообразимое программирование
```

Сразу же возникают три вопроса. Для чего могут понадобиться подобные операторы? Почему они возможны? Что они делают?

Во-первых, для написания подобного кода нет никакой разумной причины, но далеко не все коды пишутся с разумными целями. Люди, в том числе программисты, допускают ошибки. Например, если для класса `Vector` была определена операция `operator==()`, то можно ошибочно напечатать

```
if (force1 + force2 = net)
```

вместо

```
if (force1 + force2 == net)
```

Зачастую программисты стараются быть оригинальными, а это может привести к нетривиальным ошибкам.

Во-вторых, данный код допустим, поскольку конструктор копирования создает временный объект для представления возвращаемого значения. Поэтому в приведенном выше коде выражение `force1 + force2` означает такой временный объект. В операторе 1 временный объект присваивается переменной `net`. В операторах 2 и 3 переменная `net` присваивается временному объекту.

В-третьих, временный объект используется и затем отбрасывается. Например, в операторе 2 программа вычисляет сумму переменных `force1` и `force2`, копирует ответ во временный возвращаемый объект, переписывает его содержимое содержимым `net` и затем удаляет временный объект. Все исходные векторы остаются без изменений. В операторе 3 значение временного объекта выводится перед его удалением.

Если вас беспокоят возможные проблемы, обусловленные таким поведением, можете воспользоваться простым спасательным средством. Объявите возвращаемый тип константным. Например, если объявить, что операция `Vector::operator+()` возвращает тип `const Vector`, то оператор 1 остается допустимым, а операторы 2 и 3 — нет.

Итак, если метод или функция возвращает локальный объект, то должен возвращаться сам объект, а не ссылка. В данном примере программа использует конструктор копирования для создания возвращаемого объекта. Если метод или функция возвращает объект класса, для которого нет открытого конструктора копирования (например, класса `ostream`), то должна возвращаться ссылка на объект. И, наконец, некоторые методы и функции (такие как перегруженная операция присваивания) могут возвращать как объект, так и ссылку на объект. В данном примере ссылка предпочтительнее по причинам, связанным с производительностью.

Использование указателей на объекты

В программах на C++ часто применяются указатели на объекты, поэтому давайте немного попрактикуемся в этом вопросе. В листинге 12.6 используются значения индексов массива для отслеживания самой короткой строки и первой строки в алфавитном порядке. Другим примером может послужить применение указателей для указания на текущих лидеров в данных категориях. В листинге 12.7 реализован этот подход с использованием двух указателей на объекты `String`. Первоначально указатель `shortest` указывает на первый объект в массиве. Всякий раз, когда программа находит объект с более короткой строкой, она устанавливает указатель `shortest` на этот объект. Аналогично, указатель `first` отслеживает самую первую в алфавитном порядке строку. Обратите внимание, что эти два указателя не создают новые объекты, они просто указывают на существующие объекты. Поэтому они не требуют применения операции `new` для выделения дополнительной памяти.

Для разнообразия программа в листинге 12.7 использует указатель, который отслеживает новые объекты:

```
String * favorite = new String(sayings[choice]);
```

Здесь указатель `favorite` обеспечивает доступ к безымянному объекту, созданному операцией `new`. Этот синтаксис означает инициализацию нового объекта `String` с помощью объекта `sayings[choice]`. При этом вызывается конструктор копирования, поскольку тип аргумента для конструктора копирования (`const String &`) соответствует инициализирующему значению (`sayings[choice]`). Для выбора случайных значений в программе используются функции `srand()`, `rand()` и `time()`.

Листинг 12.7. `sayings2.cpp`

```
// sayings2.cpp -- использование указателей на объекты
// компилировать вместе с string1.cpp
#include <iostream>
#include <cstdlib>    // (или stdlib.h) для rand(), srand()
#include <ctime>      // (или time.h) для time()
#include "string1.h"
const int ArSize = 10;
const int MaxLen = 81;
int main()
{
   using namespace std;
   String name;
   cout <<"Hi, what's your name?\n>> ";              // ввод имени
   cin >> name;
   cout << name << ", please enter up to " << ArSize
        << " short sayings <empty line to quit>:\n";  // ввод пословиц
   String sayings[ArSize];
   char temp[MaxLen];                   // временное хранилище для строки
```

```
    int i;
    for (i = 0; i < ArSize; i++)
    {
        cout << i+1 << ": ";
        cin.get(temp, MaxLen);
        while (cin && cin.get() != '\n')
            continue;
        if (!cin || temp[0] == '\0')       // пустая строка?
            break;                         // i не инкрементируется
        else
            sayings[i] = temp;             // перегруженное присваивание
    }
    int total = i;                         // общее количество прочитанных строк
    if (total > 0)
    {
        cout << "Here are your sayings:\n"; // вывод пословиц
        for (i = 0; i < total; i++)
        cout << sayings[i] << "\n";

        // Указатели для отслеживания кратчайшей и первой строки
        String * shortest = &sayings[0];   // ициализация первым объектом
        String * first = &sayings[0];
        for (i = 1; i < total; i++)
        {
            if (sayings[i].length() < shortest->length())
                shortest = &sayings[i];
            if (sayings[i] < *first)       // сравнение значений
                first = &sayings[i];       // присваивание адреса
        }
        cout << "Shortest saying:\n" << * shortest << endl;
              // вывод кратчайшей пословицы
        cout << "First alphabetically:\n" << * first << endl;
              // вывод первой пословицы по алфавиту
        srand(time(0));
        int choice = rand() % total;       // выбор случайного индекса

        // Создание и инициализация объекта String с помощью new
        String * favorite = new String(sayings[choice]);
        cout << "My favorite saying:\n" << *favorite << endl;
              // вывод любимой пословицы
        delete favorite;
    }
    else
        cout << "Not much to say, eh?\n";  // ничего не было введено
    cout << "Bye.\n";
    return 0;
}
```

Инициализация объекта с помощью операции new

В общем случае, если *Имя_класса* — это класс, а *значение* имеет тип *Имя_типа*, то оператор

Имя_класса * pclass = new *Имя_класса*(*значение*);

вызывает следующий конструктор:

Имя_класса(*Имя_типа*);

Некоторые преобразования могут быть тривиальными, например:

Имя_класса(const *Имя типа* &);

Кроме того, если нет неоднозначности, то выполняются и обычные преобразования наподобие `int` в `double`. Инициализация в виде

Имя_класса * ptr = new *Имя_класса*;

вызывает конструктор по умолчанию.

Ниже показан пример выполнения программы из листинга 12.7:

```
Hi, what's your name?
>> Kirt Rood
Kirt Rood, please enter up to 10 short sayings <empty line to quit>:
1: a friend in need is a friend indeed
2: neither a borrower nor a lender be
3: a stitch in time saves nine
4: a niche in time saves stine
5: it takes a crook to catch a crook
6: cold hands, warm heart
7:
Here are your sayings:
a friend in need is a friend indeed
neither a borrower nor a lender be
a stitch in time saves nine
a niche in time saves stine
it takes a crook to catch a crook
cold hands, warm heart
Shortest saying:
cold hands, warm heart
First alphabetically:
a friend in need is a friend indeed
My favorite saying:
a stitch in time saves nine
Bye
```

Программа выбирает любимую поговорку случайным образом, поэтому при разных запусках будут выбираться разные поговорки даже в случае идентичных входных данных.

Повторный взгляд на операции new и delete

Обратите внимание, что программа, сгенерированная из листингов 12.4, 12.5 и 12.7, использует операции `new` и `delete` на двух уровнях. Во-первых, операция `new` используется для выделения памяти под хранение строк имен каждого создаваемого объекта. Это происходит в функциях конструктора, и потому функция-деструктор вызывает операцию `delete` для освобождения этой памяти. Поскольку каждая строка представляет собой массив символов, деструктор использует операцию `delete` со скобками. Поэтому память, используемая для хранения содержимого строк, автоматически освобождается при уничтожении объекта. Во-вторых, код в листинге 12.7 использует операцию `new` для размещения целого объекта:

```
String * favorite = new String(sayings[choice]);
```

Здесь выделяется память для хранения не строки, а объекта — т.е. для указателя `str`, который хранит адрес строки, и для члена `len`. (При этом для члена `num_strings` память не выделяется, поскольку он является статическим и хранится отдельно от объектов.)

При создании объекта вызывается конструктор, который выделяет память для хранения строки и заносит адрес строки в указатель `str`. А после завершения работы с объектом программа использует операцию `delete` для его удаления. Объект является

одиночным, поэтому в программе применяется операция `delete` без скобок — при этом освобождается только память, которая использовалась для хранения указателя `str` и члена `len`. Память, выделенная для хранения строки, на которую указывает `str`, при этом не освобождается, эту завершающую задачу выполняет деструктор (рис. 12.4).

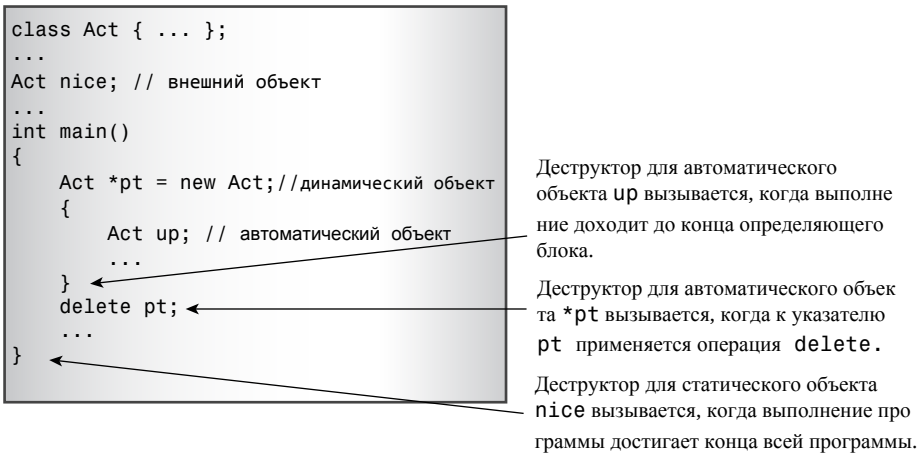


***Рис. 12.4.** Вызов деструкторов*

Деструкторы вызываются в перечисленных ниже ситуациях (рис. 12.4).

- Если объект является автоматической переменной, то деструктор объекта вызывается, когда программа завершает выполнение блока, в котором определен этот объект. Таким образом, в листинге 12.3 деструктор вызывается для `headlines[0]` и `headlines[1]`, когда выполнение покидает `main()`, а деструктор для `grub` вызывается, когда программа выходит из `callme1()`.
- Если объект является статической переменной (внешней, статической, внешней статической или из пространства имен), то его деструктор вызывается при завершении программы. Это происходит с объектом `sports` в листинге 12.3.
- Если объект создается операцией `new`, его деструктор вызывается только при явном выполнении операции `delete` для данного объекта.

Сводная информация по указателям и объектам

Относительно использования указателей на объекты должны учитываться определенные моменты (рис. 12.5).

- Указатель на объект объявляется как обычно:

```
String * glamour;
```

- Указатель можно инициализировать адресом существующего объекта:

```
String * first = &sayings[0];
```

- Указатель можно инициализировать с помощью операции `new`; при этом создается новый объект:

```
String * favorite = new String(sayings[choice]);
```

Инициализация объекта с помощью операции `new` подробно объясняется в примере на рис. 12.6.

Объявление указателя на объект класса:

```
String * glamour;
```

Инициализация указателя адресом существующего объекта:

```
String * first = &sayings[0];
```

Объект `String`

Инициализация указателя с помощью операции `new` и конструктора по умолчанию класса:

```
String * gleep = new String;
```

Инициализация указателя с помощью операции `new` и конструктора класса `String(const char*)`:

```
String * glop = new String("my my my");
```

Инициализация указателя с помощью операции `new` и конструктора класса `String(const String &)`:

```
String * favorite = new String(sayings[choice]);
```

Объект `String`

Использование операции `->` для доступа к методу класса через указатель:

```
if (sayings[i].length() < shortest->length())
```

Объект

Указатель на объект

Использование операции разыменования `(*)` для получения объекта через указатель:

```
if (sayings[i] < *first)
```

Объект

Объект

Указатель на объект

Рис. 12.5. Указатели и объекты

```
String *pveg = new String("Cabbage Heads Home");
```

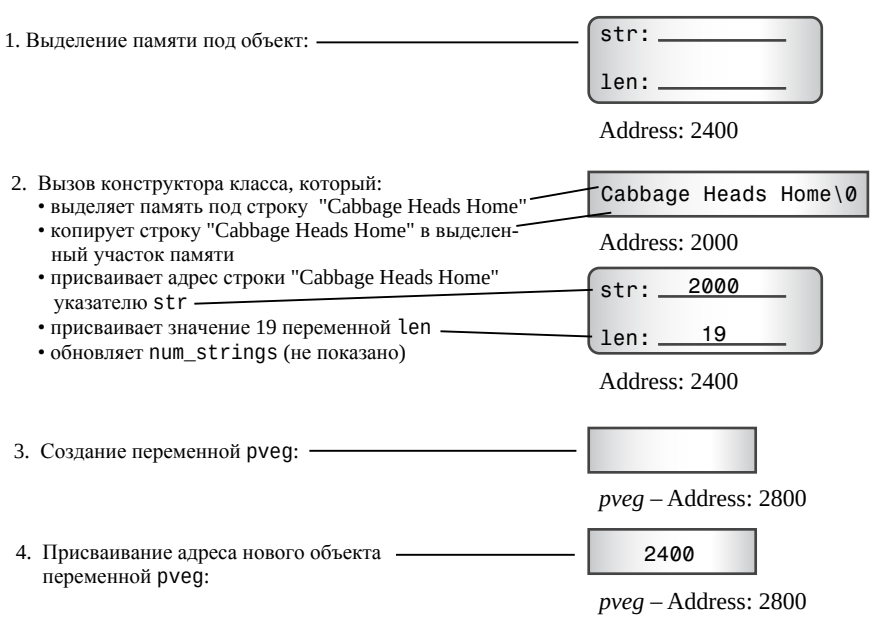


Рис. 12.6. Создание объекта с помощью операции `new`

- Использование операции new с классом вызывает соответствующий конструктор класса для инициализации вновь созданного объекта:

```
// вызов конструктора по умолчанию
String * gleep = new String;

// вызов конструктора String(const char *)
String * glop = new String("ox ox ox");

// вызов конструктора String(const String &)
String * favorite = new String(sayings[choice]);
```

- Для доступа к методу класса через указатель применяется операция ->:

```
if (sayings[i].length() < shortest->length())
```

- Для получения объекта к указателю применяется операция разыменования (*):

```
if (sayings[i] < *first)      // сравнение значений объектов
  first = &sayings[i];        // присваивание адреса объекта
```

Еще раз о new *с размещением*

Вспомните, что операция new с размещением позволяет задавать ячейки в памяти, используемые для распределения памяти. Операция new с размещением в контексте встроенных типов обсуждалась в главе 9. Использование new с размещением для объектов добавляет новые тонкости. В листинге 12.8 new с размещением используется наряду с обычной операцией new для выделения памяти под объекты. При этом определяется класс с интерактивным конструктором и деструктором, чтобы отслеживать хронологию создания и уничтожения объектов.

Листинг 12.8. placenew1.cpp

```
// placenew1.cpp -- операции new, new с размещением, но без delete
#include <iostream>
#include <string>
#include <new>
using namespace std;
const int BUF = 512;

class JustTesting
{
private:
   string words;
   int number;
public:
   JustTesting(const string & s = "Just Testing", int n = 0)
      {words = s; number = n; cout << words << " constructed\n"; }
   ~JustTesting() { cout << words << " destroyed\n";}
   void Show() const { cout << words << ", " << number << endl;}
};

int main()
{
   char * buffer = new char[BUF];                    // получение блока памяти
   JustTesting *pc1, *pc2;
   pc1 = new (buffer) JustTesting;                   // размещение объекта в buffer
   pc2 = new JustTesting("Heap1", 20);               // размещение объекта в куче
   cout << "Memory block addresses:\n" << "buffer: "
        << (void *) buffer << " heap: " << pc2 <<endl; // вывод адресов памяти
   cout << "Memory contents:\n";                     // вывод содержимого памяти
   cout << pc1 << ": ";
   pc1->Show();
```

```
    cout << pc2 << ": ";
    pc2->Show();
    JustTesting *pc3, *pc4;
    pc3 = new (buffer) JustTesting("Bad Idea", 6);
    pc4 = new JustTesting("Heap2", 10);
    cout << "Memory contents:\n";              // вывод содержимого памяти
    cout << pc3 << ": ";
    pc3->Show();
    cout << pc4 << ": ";
    pc4->Show();
    delete pc2;                                // освобождение Heap1
    delete pc4;                                // освобождение Heap2
    delete [] buffer;                          // освобождение buffer
    cout << "Done\n";
    return 0;
}
```

В программе 12.8 используется операция new для создания буфера памяти объемом 512 байт. Затем с помощью new в куче создаются два объекта типа JustTesting, и операция new с размещением пытается создать в буфере памяти два объекта типа JustTesting. В завершение программа использует операцию delete для освобождения памяти, выделенной операцией new. Ниже показан вывод программы:

```
Just Testing constructed
Heap1 constructed
Memory block addresses:
buffer: 00320AB0 heap: 00320CE0
Memory contents:
00320AB0: Just Testing, 0
00320CE0: Heap1, 20
Bad Idea constructed
Heap2 constructed
Memory contents:
00320AB0: Bad Idea, 6
00320EC8: Heap2, 10
Heap1 destroyed
Heap2 destroyed
Done
```

Как обычно, форматирование и точные значения адресов памяти могут варьироваться от системы к системе.

В листинге 12.8 имеется пара проблем с операцией new с размещением. Во-первых, при создании второго объекта new с размещением просто перезаписывает новый объект в то же место, которое уже использовано для первого объекта. Это не просто грубая ошибка, это также означает, что для первого объекта деструктор не вызывается. Конечно, такая ошибка выльется в реальные проблемы, если, например, класс использует динамическое распределение памяти для своих членов.

Во-вторых, применение операции delete для указателей pc2 и pc4 автоматически вызывает деструкторы для двух объектов, на которые указывают pc2 и pc4. Но использование операции delete [] для buffer не приводит к вызову деструкторов для объектов, созданных с помощью new с размещением.

Первый урок, который следует из этого примера — тот же, что и в главе 9. От вас зависит управление позициями памяти в буфере, который заполняется операцией new с размещением.

Для использования двух различных позиций необходимо указать два различных адреса внутри буфера, которые гарантируют, что эти позиции не перекрываются. Это можно сделать, например, так:

```
pc1 = new (buffer) JustTesting;
pc3 = new (buffer + sizeof (JustTesting)) JustTesting("Better Idea", 6);
```

Здесь указатель pc3 смещен относительно pc1 на размер объекта JustTesting.

Второй урок: если применять операцию new для хранения объектов, то для них нужно организовать и вызов деструкторов. Но как? Для объектов, созданных в куче, это можно сделать следующим образом:

```
delete pc2;  // удаление объекта, на который указывает pc2
```

Однако показанные ниже операторы использовать нельзя:

```
delete pc1;  // удаление объекта, на который указывает pc1? НЕЛЬЗЯ!
delete pc3;  // удаление объекта, на который указывает pc2? НЕЛЬЗЯ!
```

Причина состоит в том, что операция delete работает согласованно с операцией new, но не с new с размещением. Например, указатель pc3 не получает адрес, возвращаемый операцией new, поэтому delete pc3 приводит к ошибке времени выполнения. А указатель pc1 имеет то же самое числовое значение, что и buffer, но buffer инициализирован операцией new [], поэтому он должен быть освобожден операцией delete [], а не delete. Даже если buffer был инициализирован с помощью new вместо new [], операция delete pc1 освободит buffer, но не pc1. Ведь система new/delete знает о размещении 256-байтного блока, но ничего не знает о действиях new с размещением в этом блоке.

Обратите внимание на то, что программа освобождает буфер:

```
delete [] buffer;                    // освобождение buffer
```

Как гласит комментарий, оператор delete [] buffer; удаляет весь блок памяти, выделенный операцией new. Но он не вызывает деструкторы ни для одного из объектов, созданных в блоке операцией new с размещением. И действительно, интерактивные деструкторы, которые сообщают об уничтожении "Heap1" и "Heap2", молчат о "Just Testing" и "Bad Idea".

Выход из этого затруднения заключается в том, что следует явно вызывать деструктор для каждого объекта, созданного операцией new с размещением. Как правило, деструкторы вызываются автоматически, но это один из редких случаев, когда необходим явный вызов с указанием объекта, который нужно удалить. Поскольку существуют указатели на объекты, можно воспользоваться ими:

```
pc3->~JustTesting();   // уничтожение объекта, на который указывает pc3
pc1->~JustTesting();   // уничтожение объекта, на который указывает pc1
```

В листинге 12.9 устранены огрехи кода из листинга 12.8: в нем выполняется управление позициями памяти, используемыми new с размещением, а также добавлены правильные обращения к операции delete и явные вызовы деструкторов. Важно соблюдать правильный порядок удаления. Объекты, созданные операцией new с размещением, должны удаляться в порядке, обратном порядку их создания. Причина в том, что более поздний объект может зависеть от более ранних. А буфер, используемый для хранения объектов, можно освободить только после уничтожения всех содержащихся в нем объектов.

Листинг 12.9. placenew2.cpp

```
// placenew2.cpp -- операции new, new с размещением, но без delete
#include <iostream>
#include <string>
#include <new>
using namespace std;
const int BUF = 512;

class JustTesting
{
private:
   string words;
   int number;
public:
   JustTesting(const string & s = "Just Testing", int n = 0)
      {words = s; number = n; cout << words << " constructed\n"; }
   ~JustTesting() { cout << words << " destroyed\n";}
   void Show() const { cout << words << ", " << number << endl;}
};

int main()
{
   char * buffer = new char[BUF];        // получение блока памяти
   JustTesting *pc1, *pc2;
   pc1 = new (buffer) JustTesting;       // размещение объекта в buffer
   pc2 = new JustTesting("Heap1", 20);   // размещение объекта в куче
   cout << "Memory block addresses:\n" << "buffer: "
        << (void *) buffer << " heap: " << pc2 <<endl; // вывод адресов памяти
   cout << "Memory contents:\n";         // вывод содержимого памяти
   cout << pc1 << ": ";
   pc1->Show();
   cout << pc2 << ": ";
   pc2->Show();
   JustTesting *pc3, *pc4;

   // Фиксация ячейки, с которой работает new с размещением
   pc3 = new (buffer + sizeof (JustTesting))
   JustTesting("Better Idea", 6);
   pc4 = new JustTesting("Heap2", 10);
   cout << "Memory contents:\n";    // вывод содержимого памяти
   cout << pc3 << ": ";
   pc3->Show();
   cout << pc4 << ": ";
   pc4->Show();
   delete pc2;                      // освобождение Heap1
   delete pc4;                      // освобождение Heap2

   // Явное уничтожение объектов, созданных new с размещением
   pc3->~JustTesting();             // уничтожение объекта, на который указывает pc3
   pc1->~JustTesting();             // уничтожение объекта, на который указывает pc1
   delete [] buffer;                // освобождение buffer
   cout << "Done\n";
   return 0;
}
```

Ниже приведен вывод программы из листинга 12.9:

```
Just Testing constructed
Heap1 constructed
```

```
Memory block addresses:
buffer: 00320AB0 heap: 00320CE0
Memory contents:
00320AB0: Just Testing, 0
00320CE0: Heap1, 20
Better Idea constructed
Heap2 constructed
Memory contents:
00320AD0: Better Idea, 6
00320EC8: Heap2, 10
Heap1 destroyed
Heap2 destroyed
Better Idea destroyed
Just Testing destroyed
Done
```

Программа в листинге 12.9 создает два объекта с помощью new с размещением в смежных позициях и вызывает соответствующие деструкторы.

Обзор технических приемов

Вы уже сталкивались с некоторыми техническими приемами программирования, связанными с различными проблемами классов, и, возможно, вам уже трудно удерживать их в голове. В последующих разделах кратко описываются несколько таких приемов и рекомендации по их применению.

Перегрузка операции <<

Чтобы перегрузить операцию << и использовать ее для вывода в cout содержимого объекта, необходимо определить дружественную функцию операции, которая имеет следующую форму:

```
ostream & operator<<(ostream & os, const имя_класса & obj)
{
   os << ... ;  // отображение содержимого объекта
   return os;
}
```

Здесь *имя_класса* представляет собой имя класса. Если класс предоставляет общедоступные методы, которые возвращают нужное содержимое, то эти методы можно задействовать в функции операции и обойтись без дружественного статуса.

Функции преобразования

Для преобразования одиночного значения в тип класса необходимо создать конструктор класса, который имеет следующий прототип:

```
имя_класса(имя_типа value);
```

Здесь *имя_класса* представляет собой имя класса, а *имя_типа* — имя типа, который нужно преобразовать.

Для преобразования типа класса в какой-то другой тип нужно создать функцию-член класса с таким прототипом:

```
operator имя_типа();
```

Хотя данная функция не имеет объявленного типа возврата, она должна возвращать значение требуемого типа.

Используйте функции преобразования с осторожностью. При объявлении конструктора можно добавить ключевое слово `explicit`, чтобы его нельзя было задействовать для неявных преобразований.

Классы, в конструкторах которых используется операция new

Если вы разрабатываете классы, где с помощью операции `new` выделяется память, на которую указывает член класса, то следует предпринять некоторые меры предосторожности. Ранее уже была приведена сводка таких мер, но эти правила очень важно запомнить — ведь компилятор их не знает и поэтому не заметит возможных ошибок.

- К любому члену класса, который указывает на память, выделенную операцией `new`, необходимо применить операцию `delete` в деструкторе класса, чтобы освободить занимаемую память.
- Если деструктор освобождает память операцией `delete` для указателя, который является членом класса, то каждый конструктор для этого класса должен инициализировать такой указатель — с помощью либо операции `new`, либо присваивания нулевого указателя.
- Конструкторы должны содержать либо `new []`, либо `new`, но не оба варианта. Деструктор должен использовать `delete []`, если в конструкторах применяется `new []`, и `delete` — если `new`.
- Конструктор копирования должен выделять новую память, а не копировать указатель на существующую память. Это дает программе возможность инициализировать объект класса другим объектом. Конструктор, как правило, должен иметь следующий прототип:

  имяКласса`(const` *имяКласса* `&)`
- Необходимо определить функцию-член класса, которая перегружает операцию присваивания и имеет показанный ниже прототип (здесь `c_pointer` является членом класса *имя_класса* и имеет тип указателя на *имя_типа*). В следующем примере предполагается, что конструктор инициализирует переменную `c_pointer` с помощью операции `new []`:

```
имя_класса & имя_класса::operator=(const имя_класса & cn)
{
   if (this == & cn_)
      return *this; // если присваивание самому себе, то все готово
   delete [] c_pointer;
   // Определение количества единиц имя_типа, которые нужно скопировать
   c_pointer = new имя_типа[size];
   // Копирование данных, на которые указывает cn.c_pointer,
   // в позицию, указанную c_pointer
   ...
   return *this;
}
```

Моделирование очереди

А теперь применим наше расширенное знание классов к конкретной задаче. Банк "Bank of Heather" хочет открыть банкомат в супермаркете "Food Heap". Управляющий "Food Heap" переживает насчет очередей к банкомату, которые могут помешать проходу покупателей в магазине, и желает установить предельную длину очереди к банкомату. Поэтому работникам банка "Bank of Heather" необходимо оценить время ожида-

ния клиентов в очереди. Наша задача – подготовка программы, которая моделирует ситуацию, чтобы управляющий персонал магазина мог увидеть возможный эффект от установки банкомата.

Данную задачу довольно естественно представить с помощью очереди посетителей. Очередь представляет собой абстрактный тип данных (АТД), который хранит упорядоченную последовательность элементов. Новые элементы добавляются в конец очереди, а удаляются из начала. Очередь подобна стеку, только в стеке добавления и удаления выполняются с одного и того же конца. То есть стек является структурой LIFO (последним вошел – первым обслужен), а очередь – структурой FIFO (первым зашел – первым обслужен). Тип данных очереди подобен очереди в кассу или к банкомату, т.е. идеально подходит к данной задаче. Значит, в одной части проекта нужно определить класс `Queue`. (В главе 16 вы ознакомитесь с классом `queue` из стандартной библиотеки шаблонов, но гораздо полезнее разработать собственный класс, чем просто прочитать о таком классе.)

Элементами очереди являются клиенты. Представитель банка "Bank of Heather" сообщает, что в среднем треть клиентов тратит на обслуживание одну минуту, треть – две минуты и еще одна треть – три. Кроме того, клиенты появляются через случайные промежутки времени, но среднее количество клиентов в час примерно постоянно. Две другие части проекта будут посвящены разработке класса, представляющего клиентов, и сборке программы, которая моделирует отношения клиентов и очереди (рис. 12.7).

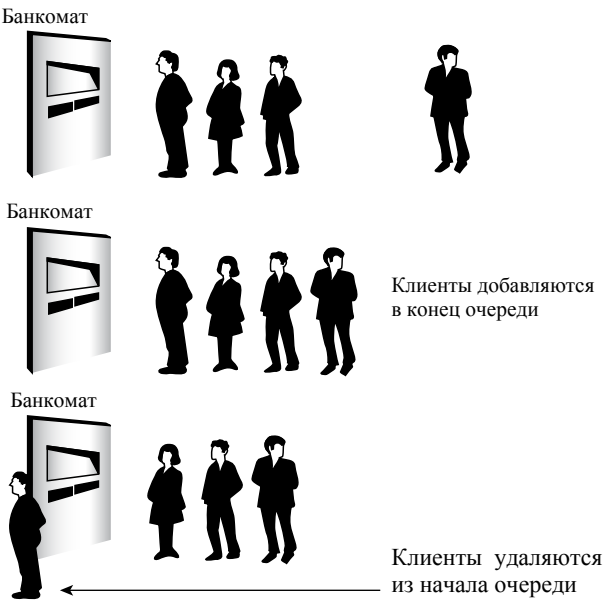


Рис. 12.7. Очередь

Класс Queue

Сначала нужно разработать класс очереди. А для этого понадобится перечислить атрибуты, которыми должен обладать требуемый вид очереди:

- очередь содержит упорядоченную последовательность элементов;
- количество элементов, которые может содержать очередь, ограничено;
- должна быть возможность создания пустой очереди;

- должна быть возможность проверки, является ли очередь пустой;
- должна быть возможность проверки, является ли очередь заполненной;
- должна быть возможность добавления элемента в конец очереди;
- должна быть возможность удаления элемента из начала очереди;
- необходима возможность определения количества элементов в очереди.

Как обычно при проектировании класса, нужно предусмотреть открытый интерфейс и закрытую реализацию.

Интерфейс класса Queue

Атрибуты очереди, перечисленные в предыдущем разделе, приводят к формированию следующего открытого интерфейса для класса `Queue`:

```
class Queue
{
   enum {Q_SIZE = 10};
private:
   // Закрытое представление будет разработано позже
public:
   Queue(int qs = Q_SIZE);     // создание очереди с предельным размером qs
   ~Queue();
   bool isempty() const;
   bool isfull() const;
   int queuecount() const;
   bool enqueue(const Item &item);     // добавление элемента в конец
   bool dequeue(Item &item);           // удаление элемента из начала
};
```

Конструктор создает пустую очередь. По умолчанию очередь может содержать до десяти элементов, но это ограничение может быть изменено с помощью аргумента при явной инициализации:

```
Queue line1;         // очередь с предельным размером 10 элементов
Queue line2(20);     // очередь с предельным размером 20 элементов
```

При использовании очереди для определения `Item` можно применять конструкцию `typedef`. (В главе 14 будет показано, как вместо этого использовать шаблоны классов.)

Реализация класса Queue

После определения интерфейса можно приступать к его реализации. Сначала необходимо решить, как представлять данные в очереди. Один из способов — применение операции `new` для динамического выделения массива с требуемым количеством элементов. Однако массивы не очень годятся для работы с очередями. Например, после удаления элемента из начала массива придется переместить каждый оставшийся элемент на одну позицию ближе к началу. Иначе придется создавать более замысловатую конструкцию, такую как циклический массив. Наиболее разумным подходом, соответствующим всем требованиям очереди, является связный список. *Связный список* состоит из последовательности узлов. Каждый *узел* содержит информацию, которую нужно хранить в списке, а также указатель на следующий узел в списке. Для очерсди в данном примере каждая часть данных имеет тип `Item`, и для представления узла можно воспользоваться следующей структурой:

```
struct Node
{
   Item item;              // данные, хранящиеся в узле
   struct Node * next;     // указатель на следующий узел
};
```

Связный список показан на рис. 12.8.

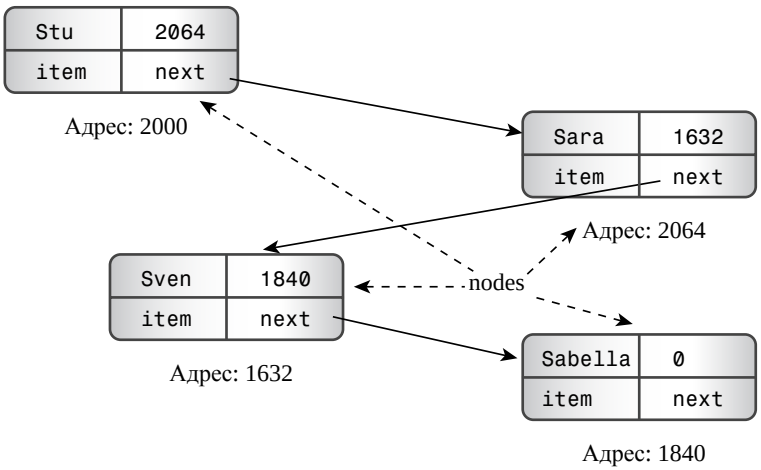


***Рис. 12.8.** Связный список*

Пример, приведенный на рис. 12.8, называется *односвязным списком*, поскольку каждый узел содержит единственную ссылку, или указатель, на другой узел. Если знать адрес первого узла, то по указателям можно пройти по всем последующим узлам в списке. Обычно в указатель внутри последнего узла заносится значение `NULL` (либо `0`), означающее, что узлов больше нет. В C++11 лучше использовать новое ключевое слово `nullptr`. Чтобы обработать связный список, нужен адрес первого узла. Для указания на начало списка можно использовать член данных класса `Queue`. В принципе, этой информации достаточно, т.к. по цепочке узлов можно добраться до любого из них. Но поскольку новый элемент всегда добавляется в конец очереди, удобно иметь также член данных, указывающий на последний узел (рис. 12.9). Кроме того, можно задействовать дополнительные данные-члены для отслеживания максимально допустимого количества элементов и текущего их количества. Тогда закрытая часть объявления класса может выглядеть следующим образом:

```
class Queue
{
private:
   // Определения области действия класса
   // Node — это вложенное определение структуры, локальное для данного класса
   struct Node { Item item; struct Node * next;};
   enum {Q_SIZE = 10};
   // 3  рытые члены класса
   Node * front;        // указатель на начало Queue
   Node * rear;         // указатель на конец Queue
   int items;           // текущее количество элементов в Queue
   const int qsize;     // максимальное количество элементов в Queue
   ...
public:
   //...
};
```

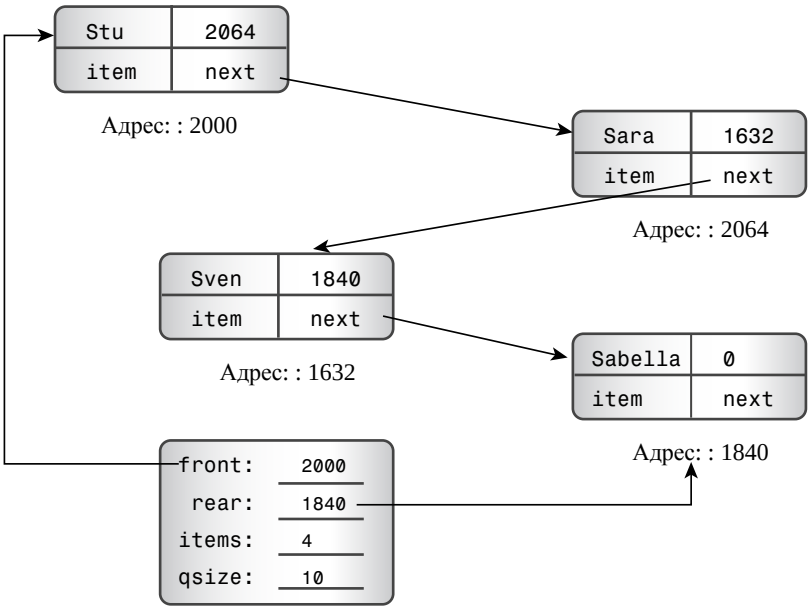


Рис. 12.9. Объект *Queue*

В приведенном объявлении используется возможность вложения в класс C++ структуры или объявления другого класса. Поскольку объявление `Node` расположено внутри класса `Queue`, его область действия ограничена этим классом. Это значит, что тип `Node` можно применять для объявления членов класса и в качестве имени типа в методах класса, но использование данного типа ограничено содержащим его классом. Это позволяет не опасаться конфликта объявления `Node` с каким-то глобальным объявлением либо с классом `Node`, объявленным внутри другого класса. Некоторые устаревшие компиляторы не поддерживают вложенные структуры и классы. Если ваш входит в их число, то структуру `Node` придется определить глобально, и областью ее действия будет весь файл.

Вложенные структуры и классы

Структура, класс или перечисление, объявленное внутри объявления класса, называется *вложенным* в класс. Областью его действия является класс. Такое объявление не создает объект данных, а задает тип, который можно использовать внутри класса. Если объявление размещено в закрытом разделе класса, то объявленный тип можно применять только внутри класса. Если объявление размещено в открытом разделе, то объявленный тип можно также использовать вне класса с помощью операции разрешения контекста. Если, например, тип `Node` объявить в открытом разделе класса `Queue`, то можно объявлять переменные типа `Queue::Node` за пределами класса `Queue`.

Итак, мы разобрались с представлением данных, и можно переходить к кодированию методов класса.

Методы класса

Конструктор класса должен предоставлять значения для членов класса. Поскольку сразу после создания очередь в данном примере должна быть пустой, для начального и конечного указателя нужно указать значения `NULL` (или `0`, или `nullptr`), а для пе-

ременной items — значение 0. Кроме того, в аргументе конструктора qs необходимо указать максимальный размер очереди qsize. Приведенная ниже реализация не работает:

```
Queue::Queue(int qs)
{
    front = rear = NULL;
    items = 0;
    qsize = qs;    // неприемлемо!
}
```

Проблема в том, что qsize — константная переменная, поэтому ее можно *инициализировать*, но ей нельзя *присвоить* некоторое значение. С концептуальной точки зрения вызов конструктора создает объект до того, как выполняется код внутри скобок. Поэтому при вызове конструктора Queue(int qs) программа сначала выделяет память для четырех переменных-членов. Затем программа начинает выполнять код в скобках и заносит значения в выделенную память с помощью обычного присваивания. Значит, если нужно инициализировать константный член данных, то это необходимо сделать, когда объект уже создан, но до того, как выполнение программы дойдет до тела конструктора. Для этого в C++ предусмотрен специальный синтаксис — *список инициализаторов членов*. Этот список состоит из инициализаторов, разделенных запятыми, с двоеточием впереди. Он помещается после закрывающей скобки списка аргументов и перед открывающей скобкой тела функции. Если член данных имеет имя mdata и его нужно инициализировать значением val, то инициализатор имеет форму mdata(val). Используя данное представление, конструктор Queue можно записать следующим образом:

```
Queue::;Queue(int qs) : qsize(qs)  // инициализация qsize значением qs
{
    front = rear = NULL;
    items = 0;
}
```

В общем случае начальное значение может быть константой или аргументом из списка аргументов конструктора. Данный прием позволяет не только инициализировать константы, и, например, конструктор Queue может выглядеть так:

```
Queue::Queue(int qs) : qsize(qs), front(NULL), rear(NULL), items(0)
{
}
```

Такой список инициализаторов может применяться только в конструкторах. Как вы уже видели, этот синтаксис необходимо использовать для константных членов класса const. Кроме того, его следует применять для членов класса, которые объявлены как ссылки:

```
class Agency {...};
class Agent
{
private:
    Agency & belong; // для инициализации нужен список инициализаторов
    ...
};
Agent::Agent(Agency & a) : belong(a) {...}
```

Это связано с тем, что ссылки, как и константные данные, могут быть инициализированы только во время создания. Для простых членов данных вроде front и items

нет особой разницы, использовать для них список инициализаторов членов или присваивание в теле функции. Однако в главе 14 будет показано, что список инициализаторов членов более эффективно применять для тех членов, которые сами являются объектами класса.

Синтаксис списка инициализаторов членов

Если, например, `Classy` — это класс, а `mem1`, `mem2` и `mem3` — данные-члены этого класса, то конструктор класса может использовать для инициализации данных-членов следующий синтаксис:

```
Classy::Classy(int n, int m) : mem1(n), mem2(0), mem3(n*m + 2)
{
    //...
}
```

Здесь для `mem1` устанавливается значение `n`, для `mem2` — значение `0`, а для `mem3` — значение `n*m + 2`. Эти инициализации производятся во время создания объекта и до того, как выполняется код в скобках. Обратите внимание на перечисленные ниже моменты.

- Такая форма может применяться только с конструкторами.
- Эту форму необходимо (по крайней мере, до C++11) использовать для инициализации нестатических константных членов данных.
- ту форму необходимо применять для инициализации ссылочных данных-членов.

Данные-члены инициализируются в том порядке, в котором они находятся в объявлении класса, а не в порядке перечисления инициализаторов.

Внимание!

Списки инициализаторов членов нельзя использовать в методах класса, отличных от конструкторов.

Форма с круглыми скобками, применяемая в списке инициализаторов членов, может применяться и при обычной инициализации. То есть при желании код

```
int games = 162;
double talk = 2.71828;
```

можно заменить кодом

```
int games(162);
double talk(2.71828);
```

При этом инициализация встроенных типов выглядит как инициализация объектов класса.

Инициализация членов внутри класса в C++11

Стандарт C++11 позволяет делать то, что выглядит вполне естественным:

```
class Classy
{
    int mem1 = 10;          // инициализация внутри класса
    const int mem2 = 20;    // инициализация внутри класса
    //...
};
```

Такой код эквивалентен использованию списка инициализации членов в конструкторах:

```
Classy::Classy() : mem1(10), mem2(20) {...}
```

Члены mem1 и mem2 инициализируются значениями 10 и 20 соответственно — если только не применяется конструктор со списком инициализаторов членов. В этом случае приведенный список переопределяет такие стандартные инициализации:

```
Classy::Classy(int n) : mem1(n) {...}
```

В такой ситуации конструктор использует значение n для инициализации mem1, а mem2 так и останется равным 20.

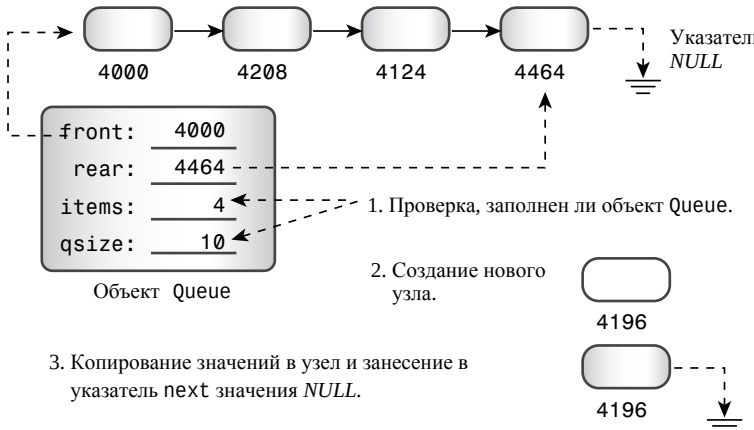
Код для функций isempty(), isfull() и queuecount() не представляет особой сложности. Если значение items равно 0, то очередь пуста. Если items равно qsize, то очередь заполнена. Возврат значения items отвечает на вопрос, сколько элементов содержится в очереди. Этот код будет приведен в листинге 12.11 далее в главе.

Добавление элемента в конец очереди выполняется несколько сложнее. Вот один из подходов:

```
bool Queue::enqueue(const Item & item)
{
  if (isfull())
     return false;
  Node * add = new Node;    // создание узла
  // При неудачном выполнении операция new генерирует исключение std::bad_alloc
  add->item = item;         // занесение указателей на узлы
  add->next = NULL;         // или nullptr
  items++;
  if (front == NULL)        // если очередь пуста,
     front = add;           // элемент помещается в начало
  else
     rear->next = add;      // иначе он помещается в конец
  rear = add;               // указатель конца указывает на новый узел
  return true;
}
```

Этот метод проходит через следующие этапы (рис. 12.10).

1. Если очередь уже полна, завершить программу. (В приведенной реализации максимальный размер задается пользователем через конструктор.)
2. Создать новый узел. Если new не может создать его, генерируется исключение (см. главу 15). Если не написан код для обработки этого исключения, программа завершается.
3. Поместить соответствующие значения в узел. В данном случае код копирует значение Item в часть данных узла и заносит в указатель следующего узла значение NULL (либо 0, либо nullptr в C++11). Это подготовка к тому, что узел будет последним элементом в очереди.
4. Увеличить счетчик элементов (items) на единицу.
5. Присоединить узел в конец очереди. Этот процесс состоит из двух частей. Во-первых, созданный узел привязывается к другим узлам в списке. Для этого в указатель next предыдущего конечного узла заносится ссылка на новый конечный узел. Во-вторых, в указатель-член rear объекта Queue заносится ссылка на новый узел, чтобы иметь доступ непосредственно к последнему узлу. Если очередь пуста, то ссылку на новый узел необходимо поместить и в указатель front. (Если в очереди всего один узел, то он является и начальным, и конечным.)



4. и 5. Изменение счетчика items, присоединение узла в конец очереди и переустановка указателя rear.

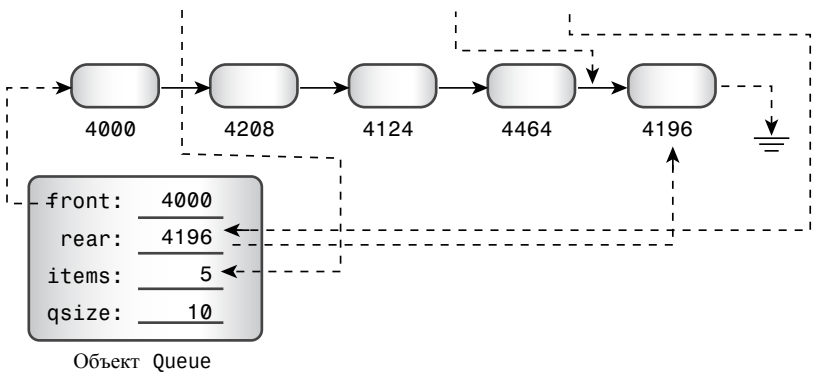


***Рис. 12.10.** Добавление элемента в конец очереди*

Удаление элемснта из начала очереди также выполняется за несколько шагов. Вот один из способов:

```
bool Queue::dequeue(Item & item)
{
    if (front == NULL)
        return false;
    item = front->item;     // в item заносится первый элемент из очереди
    items--;
    Node * temp = front;    // сохранение местоположения первого элемента
    front = front->next;    // сдвиг указателя начала на следующий элемент
    delete temp;            // удаление предыдущего первого элемента
    if (items == 0)
        rear = NULL;
    return true;
}
```

Этот метод состоит из перечисленных ниже этапов (рис. 12.11).

1. Если очередь уже пуста, завершить программу.
2. Предоставить вызывающей функции первый элемент очереди. Для этого часть данных текущего узла `front` копируется в ссылочную переменную, переданную в метод.

3. Уменьшить счетчик элементов (`items`) на единицу.
4. Сохранить расположение начального узла для последующего удаления.
5. Удалить узел из очереди. Для этого в указатель-член `front` объекта `Queue` заносится указатель на следующий узел, адрес которого находится в `front->next`.
6. Удалить предыдущий начальный узел для его повторного использования в дальнейшем.
7. Если список теперь пуст, то занести в `rear` значение `NULL`. (Начальный указатель уже равен `NULL` после установки `front->next`.) Как всегда, вместо `NULL` можно занести значение `0` или (в C++11) `nullptr`.

Шаг 4 необходим потому, что на шаге 5 очищается память, в которой находился предыдущий начальный узел.

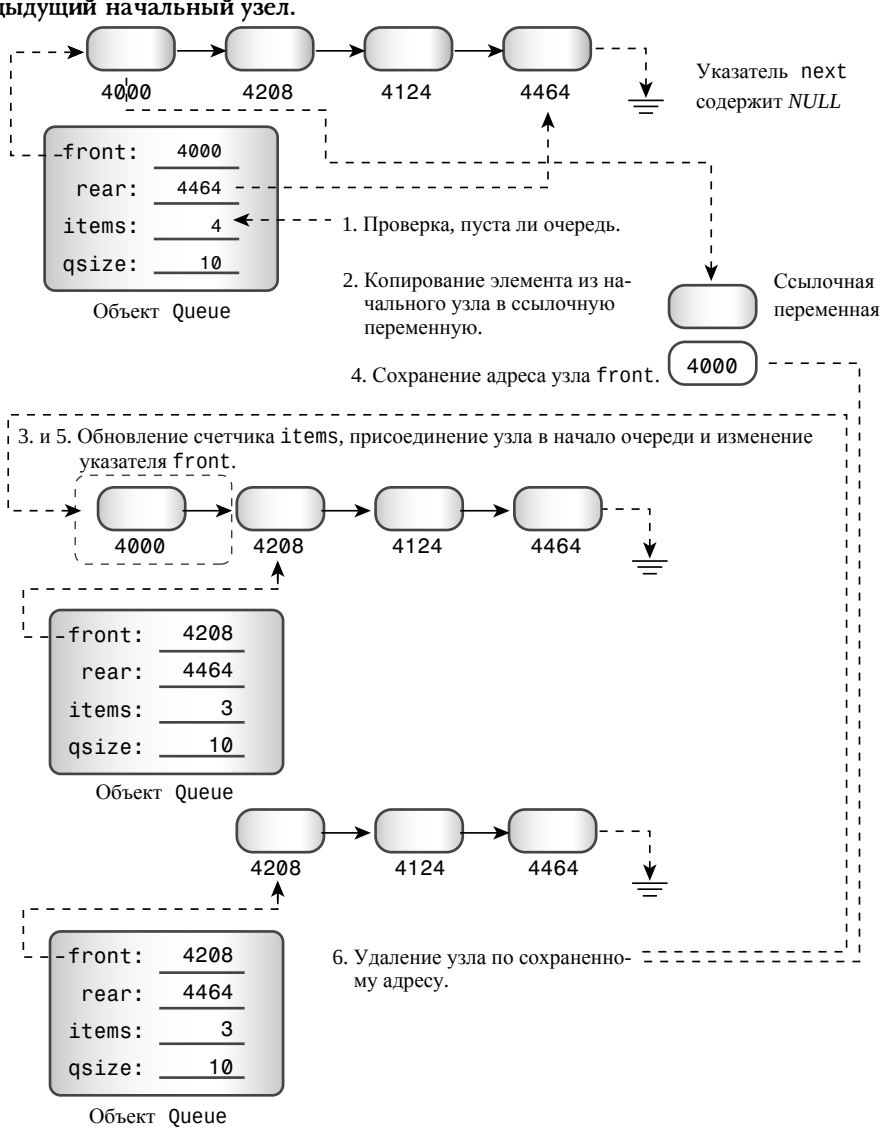


Рис. 12.11. *Удаление элемента из очереди*

Другие методы класса?

Вам необходимы еще методы? Конструктор рассматриваемого класса не использует операцию new, поэтому, на первый взгляд, может показаться, что не следует беспокоиться о специальных требованиях классов, которые применяют new в конструкторах. Конечно, это первое впечатление ошибочно – ведь при добавлении объектов в очередь выполняется операция new для создания новых узлов. Разумеется, метод dequeue() производит очистку при удалении узлов, но нет никакой гарантии, что после завершения работы очередь будет пустой. Поэтому для данного класса нужен явный деструктор – такой, который удалит все оставшиеся узлы. Ниже приведена реализация, которая поочередно удаляет все узлы, начиная с начала списка:

```
Queue::~Queue()
{
   Node * temp;
   while (front != NULL)     // пока очередь не пуста
   {
      temp = front;          // сохранение адреса начального элемента
      front = front->next;   // переустановка указателя на следующий элемент
      delete temp;           // удаление предыдущего начального элемента
   }
}
```

Мы уже знаем, что в классах, в которых используется операция new, обычно нужны явные конструкторы копирования и операции присваивания, которые выполняют глубокое копирование. Но так ли это в нашем случае? Первый вопрос, на который нужно ответить: делает ли стандартное почленное копирование то, что нам требуется? Оказывается, нет. Почленное копирование объекта Queue порождает новый объект, который указывает на начало и конец исходного связного списка. Значит, добавление элемента в копию объекта Queue изменит общий связный список. Это само по себе плохо. Но еще хуже то, что конечный указатель обновляется только в копии, т.е. с точки зрения исходного объекта список существенно искажается. Очевидно, что для клонирования или копирования очередей нужны конструктор копирования и конструктор присваивания, которые выполняют глубокое копирование.

Конечно, при этом возникает вопрос: а для чего может понадобиться копирование очереди? Например, может потребоваться сохранить снимки очереди на различных этапах моделирования. Или подать одинаковые входные данные для двух различных стратегий. В принципе, могут пригодиться операции разделения очереди, как это иногда бывает в супермаркетах при открытии дополнительной кассы. Аналогично может возникнуть необходимость в объединении двух очередей в одну или в усечении очереди.

Предположим, что в нашем моделировании такие операции не потребуются. Можно ли просто проигнорировать данные соображения и использовать уже имеющиеся методы? Конечно, можно. Однако когда-нибудь в будущем вам могут понадобиться аналогичные очереди, но с копированием. А вы можете забыть, что не написали соответствующий код копированию. В таком случае программы будут компилироваться и выполняться, но они будут выдавать загадочные результаты или просто аварийно завершаться. Поэтому лучше предусмотреть сразу конструктор копирования и операцию присваивания, даже если сейчас они не нужны.

К счастью, существует хитрый способ избежать дополнительной работы и в то же время защититься от будущих аварийных ситуаций в работе программы – определение необходимых методов как фиктивных закрытых методов:

```
class Queue
{
private:
    Queue(const Queue & q) : qsize(0) { }   // упреждающее определение
    Queue & operator=(const Queue & q) { return *this;}
    //...
};
```

Эффект здесь двоякий. Во-первых, этот код переопределяет стандартные определения методов, которые в противном случае генерируются автоматически. Во-вторых, поскольку это закрытые методы, они не могут вызываться внешним кодом. То есть если `nip` и `tuck` являются объектами `Queue`, то компилятор не разрешит следующие операторы:

```
Queue snick(nip);    // нельзя
tuck = nip;          // нельзя
```

Теперь вы не столкнетесь с загадочными проблемами во время работы программы, а получите легко отслеживаемую ошибку компилятора, гласящую, что данные методы недоступны. Этот прием полезен и при определении класса, элементы которого нельзя копировать.

В C++11 предлагается еще один способ запрета вызова методов — с помощью ключевого слова `delete`; об этом ресь пойдет в главе 18.

Есть ли еще что-то, на что следует обратить внимание? Да. Вспомните, что конструктор копирования вызывается, когда объекты передаются (или возвращаются) по значению. Однако проблемы не будет, если применять рекомендуемую практику передачи объектов как ссылок. Конструктор копирования также применяется для создания других временных объектов. Но в определении `Queue` отсутствуют операции, которые приводят к созданию временных объектов — например, перегруженная операция сложения.

Класс `Customer`

Теперь необходимо спроектировать класс клиента — `Customer`. В общем случае, клиенты банкоматов имеют много свойств, таких как имя, номер счета и баланс счета. Однако для моделирования необходимо только два свойства: момент постановки клиента в очередь и время, необходимое для выполнения клиентской транзакции. При появлении в процессе моделирования нового клиента программа должна создать новый объект клиента, сохранив в нем время появления клиента и сгенерированное случайным образом время транзакции. Когда клиент достигает начала очереди, программа должна отметить время и вычесть из него время присоединения к очереди, чтобы получить время ожидания клиента. Ниже приведен вариант определения и реализации класса `Customer`:

```
class Customer
{
private:
    long arrive;          // момент появления клиента
    int processtime;      // время обслуживания клиента
public:
    Customer() { arrive = processtime = 0; }
    void set(long when);
    long when() const { return arrive; }
    int ptime() const { return processtime; }
};
```

```
void Customer::set(long when)
{
   processtime = std::rand() % 3 + 1;
   arrive = when;
}
```

Конструктор по умолчанию создает нулевой клиент. Функция-член `set()` устанавливает для него переданное в качестве аргумента время прибытия и случайным образом выбирает значение от 1 до 3 для времени обслуживания.

В листинге 12.10 объединены объявления классов `Queue` и `Customer`, а в листинге 12.11 предоставлены их методы.

Листинг 12.10. queue.h

```
// queue.h -- интерфейс для очереди
#ifndef QUEUE_H_
#define QUEUE_H_

// Очередь, содержащая элементы Customer
class Customer
{
private:
   long arrive;          // момент появления клиента
   int processtime;      // время обслуживания клиента
public:
   Customer() { arrive = processtime = 0; }
   void set(long when);
   long when() const { return arrive; }
   int ptime() const { return processtime; }
};

typedef Customer Item;

class Queue
{
private:
   // Определения области действия класса
   // Node - вложенная структура, локальная для данного класса
   struct Node { Item item; struct Node * next;};
   enum {Q_SIZE = 10};

   // Закрытые члены класса
   Node * front;         // указатель на начало Queue
   Node * rear;          // указатель на конец Queue
   int items;            // текущее количество элементов в Queue
   const int qsize;      // максимальное количество элементов в Queue

   // Упреждающие объявления для предотвращения открытого копирования
   Queue(const Queue & q) : qsize(0) { }
   Queue & operator=(const Queue & q) { return *this; }
public:
   Queue(int qs = Q_SIZE);          // создание очереди с предельным размером qs
   ~Queue();
   bool isempty() const;
   bool isfull() const;
   int queuecount() const;
   bool enqueue(const Item &item); // добавление элемента в конец
   bool dequeue(Item &item);       // удаление элемента из начала
};
#endif
```

Листинг 12.11. queue.cpp

```
// queue.cpp -- методы классов Queue и Customer
#include "queue.h"
#include <cstdlib>              // (или stdlib.h) для rand()

// Методы класса Queue
Queue::Queue(int qs) : qsize(qs)
{
    front = rear = NULL;        // или nullptr
    items = 0;
}

Queue::~Queue()
{
    Node * temp;
    while (front != NULL)       // пока очередь не пуста
    {
        temp = front;           // сохранение адреса начального элемента
        front = front->next;    // переустановка указателя на следующий элемент
        delete temp;            // удаление предыдущего начального элемента
    }
}

bool Queue::isempty() const
{
    return items == 0;
}

bool Queue::isfull() const
{
    return items == qsize;
}

int Queue::queuecount() const
{
    return items;
}

// Добавление элемента в очередь
bool Queue::enqueue(const Item & item)
{
    if (isfull())
        return false;
    Node * add = new Node;      // создание узла

    // При неудачном выполнении операция new генерирует исключение std::bad_alloc
    add->item = item;           // занесение указателей на узлы
    add->next = NULL;           // или nullptr;
    items++;
    if (front == NULL)          // если очередь пуста,
        front = add;            // элемент помещается в начало
    else
        rear->next = add;       // иначе он помещается в конец
    rear = add;                 // указатель конца указывает на новый узел
    return true;
}

// Помещение элемента front в переменную item и его удаление из очереди
bool Queue::dequeue(Item & item)
{
    if (front == NULL)
        return false;
```

```
    item = front->item;          // в item заносится первый элемент из очереди
    items--;
    Node * temp = front;         // сохранение местоположения первого элемента
    front = front->next;         // сдвиг указателя начала на следующий элемент
    delete temp;                 // удаление предыдущего первого элемента
    if (items == 0)
        rear = NULL;
    return true;
}
// Метод класса Customer
// При появлении клиента фиксируется момент его прибытия, а время
// обслуживания выбирается случайным образом из диапазона 1-3
void Customer::set(long when)
{
    processtime = std::rand() % 3 + 1;
    arrive = when;
}
```

Моделирование работы банкомата

Теперь у нас есть все инструменты, необходимые для моделирования работы банкомата. Программа должна запрашивать у пользователя три параметра: максимальный размер очереди, количество часов, которые моделируются программой, и среднее количество клиентов в час. Затем программа должна запустить цикл, каждое выполнение которого соответствует одной минуте моделируемого времени. Во время каждого такого минутного цикла должны выполняться перечисленные ниже шаги.

1. Определить, появился ли новый клиент. Если да, то добавить клиента в очередь при условии, что для него есть место; иначе отклонить его.
2. Если никто не обслуживается, выбрать из очереди первого человека. Определить время ожидания его обслуживания и занести в счетчик `wait_time` необходимое ему время обслуживания.
3. Если в данный момент обслуживается клиент, уменьшить счетчик `wait_time` на одну минуту.
4. Вести подсчет различных параметров: количество обслуженных клиентов, количество отвергнутых клиентов, общее время, проведенное в ожидании в очереди, и общую длину очереди.

После завершения цикла моделирования программа должна выдать статистический отчет.

Интересной проблемой является определение, появился ли новый клиент. Предположим, что в среднем за час появляются 10 клиентов, т.е. один клиент каждые шесть минут. Программа вычисляет и сохраняет эту величину в переменной `min_per_cust`. Однако в реальности клиенты не будут появляться точно через 6 минут, и нужен более случайный процесс, который моделирует появление одного клиента *в среднем* за шесть минут. Для определения, появился ли клиент во время минутного цикла, в программе используется функция:

```
bool newcustomer(double x)
{
    return (std::rand() * x / RAND_MAX < 1);
}
```

Вот как работает данный код. Значение RAND_MAX определено в файле cstdlib (ранее stdlib.h) и представляет собой наибольшее значение, которое может возвращать функция rand() (наименьшее равно 0). Предположим, что среднее время x между появлениями клиентов равно 6. Тогда значение rand() * x / RAND_MAX попадает куда-то в интервал от 0 до 6. В частности, оно будет меньше 1 в среднем одну шестую часть времени. Данная функция может выдать двух клиентов с промежутком в одну минуту, а может и с интервалом 20 минут. Такое нерегулярное поведение как раз и отличает реальные процессы от хронологически точных поступлений клиентов по одному каждые 6 минут.

Данный метод не сможет работать, если среднее время между появлением клиентов меньше одной минуты, но наше моделирование и не предназначено для работы в таком напряженном режиме. Однако если понадобится эмулировать такую ситуацию, можно применить более подходящее временное разрешение — например, считать, что каждый цикл длится 10 секунд.

Детали реализации моделирования представлены в листинге 12.12. Выполнение моделирования длительного периода времени позволяет оценить долгосрочные средние величины, а моделирование коротких промежутков дает краткосрочные вариации.

Листинг 12.12. bank.cpp

```
// bank.cpp -- использование интерфейса Queue
// Компилировать вместе с queue.cpp
#include <iostream>
#include <cstdlib>   // для rand() и srand()
#include <ctime>     // для time()
#include "queue.h"
const int MIN_PER_HR = 60;

bool newcustomer(double x);                    // появился ли новый клиент?

int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   using std::ios_base;

   // Подготовка
   std::srand(std::time(0));                   // случайная инициализация rand()
   cout << "Case Study: Bank of Heather Automatic Teller\n";
   cout << "Enter maximum size of queue: ";   // ввод максимального размера очереди
   int qs;
   cin >> qs;
   Queue line(qs);                             // очередь может содержать до qs людей
   cout << "Enter the number of simulation hours: "; // ввод количества эмулируемых часов
   int hours;                                  // часы эмуляции
   cin >> hours;

   // Эмуляция будет запускать один цикл в минуту
   long cyclelimit = MIN_PER_HR * hours;       // количество циклов
   cout << "Enter the average number of customers per hour: ";
         // Ввод количества клиентов в час
   double perhour;       // среднее количество появлений за час
   cin >> perhour;
   double min_per_cust;  // среднее время между появлениями
   min_per_cust = MIN_PER_HR / perhour;
   Item temp;                                  // данные нового клиента
   long turnaways = 0;                         // не допущены в полную очередь
```

```
   long customers = 0;                          // присоединены к очереди
   long served = 0;                             // обслужены во время эмуляции
   long sum_line = 0;                           // общая длина очереди
   int wait_time = 0;                           // время до освобождения банкомата
   long line_wait = 0;                          // общее время в очереди

   // Запуск моделирования
   for (int cycle = 0; cycle < cyclelimit; cycle++)
   {
      if (newcustomer(min_per_cust))            // есть подошедший клиент
      {
         if (line.isfull())
            turnaways++;
         else
         {
            customers++;
            temp.set(cycle);                    // cycle = время прибытия
            line.enqueue(temp);                 // добавление новичка в очередь
         }
      }

      if (wait_time <= 0 && !line.isempty())
      {
         line.dequeue (temp);                   // обслуживание следующего клиента
         wait time = temp.ptime();              // в течение wait_time минут
         line wait += cycle - temp.when();
         served++;
      }

      if (wait time > 0)
         wait time--;
      sum_line += line.queuecount();
   }

   // Вывод результатов
   if (customers > 0)
   {
      cout << "customers accepted: " << customers << endl;   // принято клиентов
      cout << " customers served: " << served << endl;       // обслужено клиентов
      cout << " turnaways: " << turnaways << endl;           // не принято клиентов
      cout << "average queue size: ";                        // средний размер очереди
      cout.precision(2);
      cout.setf(ios_base::fixed, ios_base::floatfield);
      cout << (double) sum_line / cyclelimit << endl;
      cout << " average wait time: "            // среднее время ожидания (минут)
           << (double) line_wait / served << " minutes\n";
   }
   else
      cout << "No customers!\n";                             // клиентов нет

   cout << "Done!\n";
   return 0;
}

// x = среднее время в минутах между клиентами
// возвращается значение true, если в эту минуту появляется клиент
bool newcustomer(double x)
{
   return (std::rand() * x / RAND_MAX < 1);
}
```

На заметку!

Ваш компилятор может не воспринимать тип `bool`. В таком случае можно использовать `int` вместо `bool` — значение `0` вместо `false` и `1` вместо `true`. Возможно, также понадобится включить библиотеки `stdlib.h` и `time.h` вместо более новых `cstdlib` и `ctime`. Кроме того, может потребоваться самостоятельно определить `RAND_MAX`.

Ниже показано несколько примеров выполнения программы из листингов 12.10, 12.11 и 12.12:

```
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 15
customers accepted: 1485
  customers served: 1485
        turnaways: 0
average queue size: 0.15
 average wait time: 0.63 minutes
Done!
```

```
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 30
customers accepted: 2896
  customers served: 2888
        turnaways: 101
average queue size: 4.64
 average wait time: 9.63 minutes
Done!
```

```
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 20
Enter the number of simulation hours: 100
Enter the average number of customers per hour: 30
customers accepted: 2943
  customers served: 2943
        turnaways: 93
average queue size: 13.06
 average wait time: 26.63 minutes
Done!
```

Обратите внимание, что переход от 15 к 30 клиентам в час не удваивает среднее время ожидания, а увеличивает примерно в 15 раз. Увеличение максимальной длины очереди лишь ухудшает ситуацию. Однако при эмуляции не учитывается то, что многие клиенты, раздраженные долгим ожиданием, могут просто покинуть очередь.

Ниже приводится еще несколько примеров запуска программы из листинга 12.12; они демонстрируют возможные кратковременные вариации, даже если среднее количество клиентов за час остается постоянным:

```
Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 114
  customers served: 110
        turnaways: 0
```

```
average queue size: 2.15
 average wait time: 4.52 minutes
Done!

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 121
  customers served: 116
       turnaways: 5
average queue size: 5.28
 average wait time: 10.72 minutes
Done!

Case Study: Bank of Heather Automatic Teller
Enter maximum size of queue: 10
Enter the number of simulation hours: 4
Enter the average number of customers per hour: 30
customers accepted: 112
  customers served: 109
       turnaways: 0
average queue size: 2.41
 average wait time: 5.16 minutes
Done!
```

Резюме

В этой главе были рассмотрены многие важные аспекты определения и использования классов. Некоторые из этих аспектов представляют собой тонкие — и даже трудные — концепции. Если какие-то из них показались вам непонятными или слишком сложными, не огорчайтесь: так бывает почти у всех начинающих программистов на C++. Зачастую единственный путь познания действительно ценных концепций — таких как конструкторы копирования — это решение проблем, возникающих по причине их игнорирования. Поэтому кое-что из материала данной главы может быть неясным, пока вы не разберетесь самостоятельно.

Операцию `new` можно использовать в конструкторе класса, чтобы выделить память под данные, а затем присвоить адрес этой памяти какому-то члену класса. Это позволяет классу, например, работать со строками различных размеров без жесткого кодирования заранее установленного размера. Но операция `new` в конструкторах класса может стать и источником проблем во время прекращения существования объекта. Если объект имеет указатели-члены, которые указывают на память, выделенную операцией `new`, то освобождение памяти, которую занимал объект, не освобождает автоматически память, на которую указывают указатели-члены этого объекта. Поэтому если в конструкторе класса выделяется память с помощью операции `new`, то в деструкторе класса необходимо использовать операцию `delete` для освобождения этой памяти. Тогда при уничтожении объекта автоматически запускается и удаление указываемой памяти.

Кроме того, объекты с членами, которые указывают на память, распределенную операцией `new`, являются источником сложностей при инициализации одного объекта другим либо при присваивании одного объекта другому. По умолчанию в C++ применяются почленная инициализация и присваивание, когда полученные объекты являются точными копиями членов исходных объектов. Если исходный член указывает на блок данных, то и член в копии указывает на тот же самый блок. Если программа в конце работы удаляет два объекта, деструктор класса пытается удалить один и тот

же блок дважды, что является ошибкой. Выходом служит определение специального конструктора копирования, который переопределяет инициализацию, и перегруженной операции присваивания. В любом случае новое определение должно создавать копии всех данных, на которые имеются указатели, чтобы новый объект указывал на эти копии. Тогда старый и новый объекты будут указывать на отдельные (хотя и одинаковые) данные, не перекрывающие друг друга. Те же рассуждения применяются и к операции присваивания. Во всех случаях требуется создание глубокой копии — т.е. необходимо копировать сами данные, а не только указатели на них.

Если объект имеет автоматическую или внешнюю память, то деструктор для этого объекта вызывается автоматически, когда объект прекращает свое существование. Если память для объекта выделяется операцией `new`, и его адрес присваивается указателю, то деструктор для данного объекта вызывается автоматически при применении операции `delete` к указателю. Однако если память для объектов класса выделять с помощью операции `new` с размещением, а не обычной `new`, то на программиста возлагается ответственность за явный вызов деструктора с указателем на такой объект. C++ позволяет помещать определения структур, классов и перечислений внутри класса. Подобные вложенные типы действительны только в пределах класса — т.е. локальны для класса — и не конфликтуют со структурами, классами и перечислениями с таким же именем, определенным где-либо еще.

В C++ предусмотрен специальный синтаксис для конструкторов класса, которые могут применяться для инициализации данных-членов. Это двоеточие, за которым следует список инициализаторов, разделенных запятыми. Такой список размещается между закрывающей круглой скобкой аргументов конструктора и открывающей фигурной скобкой тела функции. Каждый инициализатор состоит из имени инициализируемого члена, за которым следует начальное значение в круглых скобках. Такие инициализации применяются во время создания объекта и до операторов в теле функции. Синтаксис выглядит следующим образом:

```
queue(int qs) : qsize(qs), items(0), front(NULL), rear(NULL) { }
```

Данная форма обязательна, если член данных является нестатическим константным членом или ссылкой, за исключением того, что инициализация внутри класса, доступная в C++11, может использоваться для нестатических константных членов.

C++11 позволяет осуществлять инициализацию внутри класса (т.е. инициализацию в определении класса):

```
class Queue
{
private:
    ...
    Node * front = NULL;
    enum {Q_SIZE = 10};
    Node * rear = NULL;
    int items = 0;
    const int qsize = Q_SIZE;
    ...
};
```

Этот способ эквивалентен применению списка инициализаторов членов. Однако любой конструктор, использующий список инициализаторов членов, переопределяет соответствующие инициализации внутри класса.

Как вы уже поняли, классы требуют гораздо больше осторожности и внимания к деталям, нежели простые структуры C-стиля. С другой стороны, польза от них перевешивает такие неудобства.

Вопросы для самоконтроля

1. Пусть класс `String` содержит следующие закрытые члены:

```
class String
{
private:
   char * str;  // указывает на строку, распределенную операцией new
   int len;     // хранит длину строки
   //...
};
```

   а. Что неправильно в следующем конструкторе по умолчанию?

```
String::String() {}
```

   б. Что неправильно в следующем конструкторе?

```
String::String(const char * s)
{
   str = s;
   len = strlen(s);
}
```

   в. Что неправильно в следующем конструкторе?

```
String::String(const char * s)
{
   strcpy(str, s);
   len = strlen(s);
}
```

2. Назовите три проблемы, которые могут возникнуть при определении класса, в котором указатель-член инициализируется с помощью операции `new`. Укажите, как их можно устранить.

3. Какие методы класса компилятор генерирует автоматически, если они не представлены явно? Опишите, как ведут себя эти неявно сгенерированные функции.

4. Найдите и исправьте ошибки в следующем объявлении класса:

```
class nifty
{
// Данные
   char personality[];
   int talents;
// Методы
   nifty();
   nifty(char * s);
   ostream & operator<<(ostream & os, nifty & n);
}
nifty:nifty()
{
   personality = NULL;
   talents = 0;
}
nifty:nifty(char * s)
{
   personality = new char [strlen(s)];
   personality = s;
   talents = 0;
}
```

```
ostream & nifty:operator<<(ostream & os, nifty & n)
{
   os << n;
}
```

5. Имеется следующее объявление класса:

```
class Golfer
{
private:
   char * fullname;  // указывает на строку, содержащую имя игрока в гольф
   int games;        // хранит количество сыгранных игр
   int * scores;     // указывает на первый элемент массива счетов игр
public:
   Golfer();
   Golfer(const char * name, int g = 0);
   // Создает пустой динамический массив из g элементов, если g > 0
   Golfer(const Golfer & g);
   ~Golfer();
};
```

а. Какие методы класса будут вызываться следующими операторами?

```
Golfer nancy;                    // #1
Golfer lulu("Little Lulu");      // #2
Golfer roy("Roy Hobbs", 12);     // #3
Golfer * par = new Golfer;       // #4
Golfer next = lulu;              // #5
Golfer hazzard = "Weed Thwacker"; // #6
*par = nancy;                    // #7
nancy = "Nancy Putter";          // #8
```

б. Ясно, что классу требуется больше методов для того, чтобы он был действительно полезным. Какой дополнительный метод нужен для защиты данных от разрушения?

Упражнения по программированию

1. Имеется следующее объявление класса:

```
class Cow {
   char name[20];
   char * hobby;
   double weight;
public:
   Cow();
   Cow(const char * nm, const char * ho, double wt);
   Cow(const Cow c&);
   ~Cow();
   Cow & operator=(const Cow & c);
   void ShowCow() const;  // отображение всех данных cow
};
```

Напишите реализацию для этого класса и короткую программу, использующую все функции-члены.

2. Усовершенствуйте объявление класса `String` (т.е. замените `string1.h` на `string2.h`), выполнив перечисленные ниже действия.

а. Перегрузите операцию + для объединения двух строк в одну.

б. Напишите функцию-член `stringlow()`, которая преобразует все буквенные символы в строке в нижний регистр. (Не забудьте о семействе `cctype` символьных функций.)

в. Напишите функцию-член `stringup()`, которая преобразует все буквенные символы в строке в верхний регистр.

г. Напишите функцию-член, которая принимает аргумент типа `char` и возвращает количество раз, которое символ появляется в строке.

Проверьте работу полученного класса в следующей программе:

```
// pe12_2.cpp
#include <iostream>
using namespace std;
#include "string2.h"
int main()
{
   String s1(" and I am a C++ student.");
   String s2 = "Please enter your name: "; // ввод имени
   String s3;
   cout << s2;                    // перегруженная операция <<
   cin >> s3;                     // перегруженная операция >>
   s2 = "My name is " + s3;       // перегруженные операции =, +
   cout << s2 << ".\n";
   s2 += s2 + s1;
   s2.stringup();                 // преобразование строки в верхний регистр
   cout << "The string\n" << s2 << "\ncontains " << s2.has('A')
        << " 'A' characters in it.\n";
   s1 = "red";                    // String(const char *),
                                  // тогда String & operator=(const String&)
   String rgb[3] = { String(s1), String("green"), String("blue")};
   cout << "Enter the name of a primary color for mixing light: "; // ввод цвета
   String ans;
   bool success = false;
   while (cin >> ans)
   {
      ans.stringlow();            // преобразование строки в нижний регистр
      for (int i = 0; i < 3; i++)
      {
         if (ans == rgb[i])       // перегруженная операция ==
         {
            cout << "That's right!\n";
            success = true;
            break;
         }
      }
      if (success)
         break;
      else
         cout << "Try again!\n";
   }
   cout << "Bye\n";
   return 0;
}
```

Вывод программы должен выглядеть приблизительно так:

```
Please enter your name: Fretta Farbo
My name is Fretta Farbo.
The string
```

```
MY NAME IS FRETTA FARBO AND I AM A C++ STUDENT.
contains 6 'A' characters in it.
Enter the name of a primary color for mixing light: yellow
Try again!
BLUE
That's right!
Bye
```

3. Перепишите класс `Stock`, описанный в листингах 10.7 и 10.8 в главе 10, чтобы он использовал для хранения названий пакетов акций непосредственно динамически выделенную память, а не объекты класса `string`. Кроме того, замените функцию-член `show()` перегруженным определением `operator<<()`. Протестируйте новое определение с помощью программы из листинга 10.9.

4. Имеется следующий вариант класса `Stack`, определенного в листинге 10.10:

```
// stack.h -- объявление класса для АТД стека
typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};                // константа, специфичная для класса
    Item * pitems;                  // хранит элементы стека
    int size;                       // количество элементов в стеке
    int top;                        // индекс для верхнего элемента стека
public:
    Stack(int n = 10);              // создает стек с n элементами
    Stack(const Stack & st);
    ~Stack();
    bool isempty() const;
    bool isfull() const;
    // push() возвращает значение false, если стек уже полный,
    // и true в противном случае
    bool push(const Item & item);   // добавление элемента в стек
    // pop() возвращает значение false, если стек уже пустой,
    // и true в противном случае
    bool pop(Item & item);          // извлечение элемента из стека
    Stack & operator=(const Stack & st);
};
```

Как понятно из закрытых членов, данный класс использует динамически выделенный массив для хранения элементов стека. Перепишите методы для соответствия новому представлению и напишите программу, которая демонстрирует работу всех методов, включая конструктор копирования и операцию присваивания.

5. Исследование банка "Bank of Heather" показало, что клиенты банкомата не ожидают в очереди более одной минуты. С помощью модели из листинга 12.10 найдите количество клиентов за час, которое приводит к среднему времени ожидания, равному одной минуте. (Применяйте по меньшей мере 100-часовый период моделирования.)

6. Банк "Bank of Heather" интересуется, что произойдет, если установить второй банкомат. Модифицируйте код моделирования в данной главе так, чтобы поддерживались две очереди. Сделайте так, чтобы клиент присоединялся к первой очереди, если в ней меньше людей, и ко второй — в противном случае. Найдите количество клиентов за час, которое приводит к среднему времени ожидания, равному одной минуте. (Обратите внимание, что это — нелинейная задача, т.е. удвоение количества банкоматов не удваивает количество клиентов, которые могут быть обслужены за час с максимальным ожиданием в одну минуту.)

13

Наследование классов

В ЭТОЙ ГЛАВЕ...

- Наследование как отношение *является*
- Открытое порождение одного класса от другого
- Защищенный доступ
- Списки инициализаторов членов в конструкторах
- Повышающее и понижающее приведение типа
- Виртуальные функции-члены
- Раннее (статическое) связывание и позднее (динамическое) связывание
- Абстрактные базовые классы
- Чистые виртуальные функции
- Когда и как использовать открытое наследование

Одна из главных целей объектно-ориентированного программирования — повторное использование кода. При разработке нового проекта, особенно крупного, удобнее повторно использовать уже проверенный код, а не заново изобретать его. Применение старого кода экономит время, а поскольку он уже использован и проверен, в программу не будут внесены новые ошибки. К тому же чем меньше приходится заниматься мелкими деталями, тем удобнее сосредоточиться на общей стратегии программы.

Традиционные библиотеки функций C позволяют многократно использовать в программах стандартные, предварительно скомпилированные функции, такие как `strlen()` и `rand()`. Многие поставщики разрабатывают специализированные библиотеки, которые расширяют стандартную библиотеку C. Например, можно приобрести библиотеки функций управления базами данных и функций управления изображением на экране. Однако с библиотеками функций связано ограничение: если поставщик не предоставляет исходный код для своих библиотек (а чаще всего это так), то вы не сможете расширить или изменить функции в соответствии со своими конкретными потребностями. Вместо этого приходится формировать программу так, чтобы подстроиться под библиотеку. Даже если поставщик передает исходный код, при его модификации можно непреднамеренно изменить другие части функции или отношения между функциями библиотеки.

Классы C++ обеспечивают более высокий уровень повторного использования кода. Многие поставщики сейчас предлагают библиотеки, которые состоят из объявлений и реализаций классов. Поскольку класс объединяет представление данных с методами, образуется более интегрированный пакет, чем библиотека функций. К примеру, единственный класс может предоставлять все средства для управления диалоговыми окнами. Часто для библиотек классов доступен исходный код, и каждый может модифицировать их в соответствии со своими потребностями. Однако в C++ для расширения и изменения классов имеется более удобный метод, чем правка кода. Этот способ — *наследование классов* — позволяет порождать новые классы от старых, называемых *базовыми классами*. Производный класс наследует все свойства, включая методы, старого класса. Унаследовать состояние обычно легче, чем построить его с нуля. Точно так же порождение класса с помощью наследования обычно проще разработки нового. Ниже перечислено, что позволяет делать наследование.

- Добавлять новые возможности в существующий класс. Например, в существующий базовый класс массива можно добавить арифметические операции.
- Добавлять данные, которые представляет класс. Например, взяв за основу базовый класс строки, можно породить класс, в котором добавлен член данных, представляющий цвет, и который будет использоваться при выводе строки на экран.
- Изменять поведение методов класса. Например, от класса `Passenger`, который представляет услуги, предоставляемые пассажиру авиалинии, можно породить класс `FirstClassPassenger` с более высоким уровнем обслуживания.

Конечно, всего этого можно добиться, скопировав исходный код класса и изменив его, но механизм наследования позволяет просто добавлять новые возможности. Для создания производного класса даже не нужен доступ к исходному коду. Так что если вы приобрели библиотеку классов, которая содержит только заголовочные файлы и скомпилированный код для методов класса, вы все равно сможете порождать новые классы от библиотечных классов. Вы можете даже передавать собственные классы другим пользователям, не раскрывая деталей реализации, но предоставляя им возможность добавления свойств к этим классам.

Наследование — превосходная концепция, и его основная реализация достаточно проста. Однако управление наследованием так, чтобы оно работало должным образом во всех ситуациях, требует некоторых уточнений. В данной главе рассматриваются как простые, так и более сложные аспекты наследования.

Начало: простой базовый класс

Когда один класс наследуется от другого, исходный класс называется *базовым классом*, а наследующий — *производным классом*. Чтобы проиллюстрировать прием наследования, начнем с разработки базового класса. Предположим, клуб "Webtown Social Club" решил вести учет своих членов, играющих в настольный теннис. Как ведущий программист клуба, вы написали простой класс `TableTennisPlayer`, который показан в листингах 13.1 и 13.2.

Листинг 13.1. `tabtenn0.h`

```
// tabtenn0.h -- базовый класс для клуба по настольному теннису
#ifndef TABTENN0_H_
#define TABTENN0_H_
#include <string>
using std::string;

// Простой базовый класс
class TableTennisPlayer
{
private:
   string firstname;
   string lastname;
   bool hasTable;
public:
   TableTennisPlayer (const string & fn = "none",
                      const string & ln = "none", bool ht = false);
   void Name() const;
   bool HasTable() const { return hasTable; };
   void ResetTable(bool v) { hasTable = v; };
};
#endif
```

Листинг 13.2. `tabtenn0.cpp`

```
//tabtenn0.cpp -- методы простого базового класса
#include "tabtenn0.h"
#include <iostream>

TableTennisPlayer::TableTennisPlayer (const string & fn,
   const string & ln, bool ht) : firstname(fn),
   lastname(ln), hasTable(ht) {}

void TableTennisPlayer::Name() const
{
   std::cout << lastname << ", " << firstname;
}
```

Класс `TableTennisPlayer` просто содержит имена игроков, а также наличие у них столов. Здесь стоит отметить пару моментов. Во-первых, для хранения имен в классе используется стандартный класс `string`. Он удобнее, гибче и надежнее, чем символьный массив. И он профессиональнее класса `String` из главы 12.

Во-вторых, в конструкторе задействован список инициализаторов членов. Инициализацию можно выполнить и так:

```
TableTennisPlayer::TableTennisPlayer (const string & fn,
                                      const string & ln, bool ht)
{
    firstname = fn;
    lastname = ln;
    hasTable = ht;
}
```

Правда, при таком подходе сначала вызывается конструктор `string` по умолчанию для `firstname`, а затем выполняется операция присваивания для `string`, которая заносит в `firstname` значение `fn`. Синтаксис списка инициализаторов членов экономит один шаг: он просто инициализирует член `firstname` значением `fn` с помощью конструктора копирования `string`.

Код в листинге 13.3 демонстрирует этот скромный класс в действии.

Листинг 13.3. `usett0.cpp`

```
// usett0.cpp -- использование базового класса
#include <iostream>
#include "tabtenn0.h"

int main ( void )
{
    using std::cout;
    TableTennisPlayer player1("Chuck", "Blizzard", true);
    TableTennisPlayer player2("Tara", "Boomdea", false);
    player1.Name();
    if (player1.HasTable())
        cout << ": has a table.\n";
    else
        cout << ": hasn't a table.\n";
    player2.Name();
    if (player2.HasTable())
        cout << ": has a table";
    else
        cout << ": hasn't a table.\n";
    return 0;
}
```

Ниже показан вывод программы из листингов 13.1, 13.2 и 13.3:

```
Blizzard, Chuck: has a table.
Boomdea, Tara: hasn't a table.
```

Обратите внимание, что в программе используются конструкторы с аргументами в виде строк стиля C:

```
TableTennisPlayer player1("Чак", "Близзард", true);
TableTennisPlayer player2("Тара", "Бумди", false);
```

Однако формальные параметры конструктора объявлены как `const string &`. Типы не совпадают, но у класса `string`, почти как у класса `String` из главы 12, имеется конструктор с параметром `const char *`, который автоматически вызывается для создания объекта `string`, инициализированного строкой в стиле C. Короче говоря, в качестве аргумента конструктора `TableTennisPlayer` можно использовать

как объект string, так и строку в стиле C. В первом случае вызывается конструктор string с параметром const string &, а во втором — конструктор string с параметром const string *.

Порождение класса

Некоторые члены клуба "Webtown Social Club" принимали участие в местных турнирах по настольному теннису, и для них требуется класс, который содержит рейтинговые баллы, заработанные ими в этих играх. Можно не начинать с пустого места, а породить новый класс от TableTennisPlayer. Первым делом, объявление класса RatedPlayer должно отражать, что он порожден от TableTennisPlayer:

```
// RatedPlayer порожден от базового класса TableTennisPlayer
class RatedPlayer : public TableTennisPlayer
{
   ...
};
```

Здесь двоеточие указывает на то, что класс RatedPlayer основан на классе TableTennisPlayer.

Данный конкретный заголовок означает, что TableTennisPlayer является общедоступным базовым классом — это называется *открытым порождением*. Объект производного класса содержит в себе объект базового класса. При открытом порождении открытые члены базового класса становятся открытыми членами производного класса. Закрытые порции базового класса становятся частью производного класса, однако доступ к ним возможен только через открытые и защищенные методы базового класса. (Защищенные члены будут рассмотрены ниже.)

Что при этом происходит? Если объявить объект RatedPlayer, он будет обладать следующими особыми характеристиками.

- Объект производного типа хранит данные-члены базового типа. (Производный класс наследует реализацию базового класса.)
- Объект производного типа может использовать методы базового типа. (Производный класс наследует интерфейс базового класса.)

Таким образом, объект RatedPlayer может хранить имя и фамилию каждого игрока, а также сведения о том, имеет ли игрок стол. Также объект RatedPlayer может использовать методы Name(), HasTable() и ResetTable() из класса TableTennisPlayer (рис. 13.1).

Что необходимо добавить к этим унаследованным свойствам?

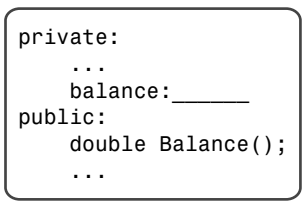
- Производному классу нужны собственные конструкторы.
- Производный класс может при необходимости добавлять дополнительные данные-члены и методы.

В нашем случае классу требуется еще один член данных ratings для хранения рейтинга. В нем также нужен метод для выборки и очистки рейтинга. Поэтому объявление класса может выглядеть следующим образом:

```
// простой производный класс
class RatedPlayer : public TableTennisPlayer
{
private:
   unsigned int rating;    // добавленный член данных
```

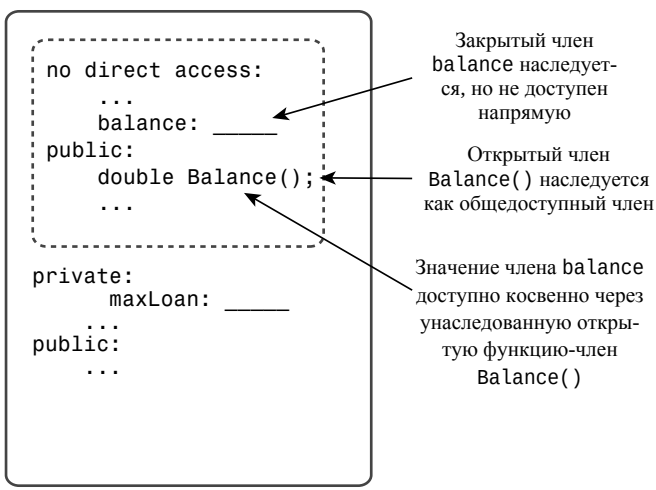
```
public:
   RatedPlayer (unsigned int r = 0, const string & fn = "none",
                const string & ln = "none", bool ht = false);
   RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
   unsigned int Rating() const { return rating; }   // добавленный метод
   void ResetRating (unsigned int r) {rating = r;}  // добавленный метод
};
```

Конструкторы должны предоставлять данные для новых членов, если они есть, а также для унаследованных членов. Первый конструктор `RatedPlayer` использует отдельные формальные параметры для каждого члена, а второй конструктор — параметр `TableTennisPlayer`, связывающий три элемента (`firstname`, `lastname` и `hasTable`) в единое целое.



Объект `BankAccount`

```
class Overdraft : public BankAccount {...};
```



Объект `Overdraft`

***Рис. 13.1.** Объекты базового и производного классов*

Конструкторы: варианты доступа

Производный класс не имеет непосредственного доступа к закрытым членам базового класса, и он вынужден обращаться к ним с помощью методов базового класса. Например, конструкторы `RatedPlayer` не могут непосредственно устанавливать значения унаследованных членов (`firstname`, `lastname` и `hasTable`). Поэтому, чтобы получить доступ к закрытым членам базового класса, они должны использовать

открытые методы базового класса. В частности, конструкторы производного класса должны использовать конструкторы базового класса.

Когда программа создает объект производного класса, сначала конструируется объект базового класса. Это означает, что объект базового класса должен быть создан до того, как программа войдет в тело конструктора производного класса. Для этого в C++ применяются списки инициализаторов членов. Вот, к примеру, код первого конструктора `RatedPlayer`:

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
    const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
   rating = r;
}
```

Выражение

```
: TableTennisPlayer(fn, ln, ht)
```

является списком инициализаторов членов. Это исполняемый код, и он вызывает конструктор `TableTennisPlayer`. Предположим, например, что программа содержит следующее объявление:

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
```

Конструктор `RatedPlayer` присваивает формальным параметрам `fn`, `ln` и `ht` фактические аргументы `"Mallory"`, `"Duck"` и `true`. Потом он передает эти параметры как фактические аргументы конструктору `TableTennisPlayer`. Этот конструктор, в свою очередь, создает вложенный объект `TableTennisPlayer` и сохраняет в нем данные `"Mallory"`, `"Duck"` и `true`. Затем программа входит в тело конструктора `RatedPlayer`, завершает создание объекта `RatedPlayer` и присваивает члену `rating` значение параметра `r` — т.е. `1140` (еще один пример показан на рис. 13.2).

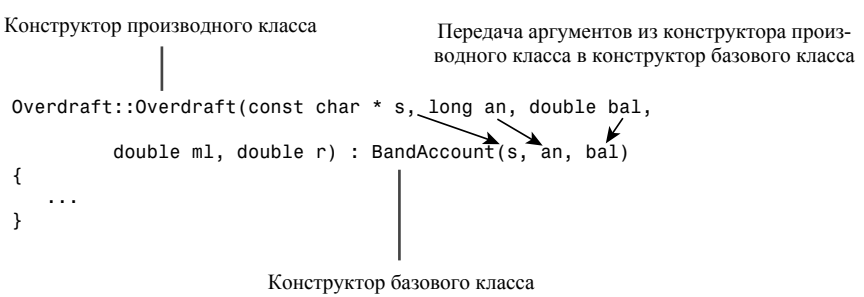


Рис. 13.2. Передача аргументов конструктору базового класса

Что произойдет, если опустить список инициализаторов членов?

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
    const string & ln, bool ht) // что будет без списка инициализаторов?
{
   rating = r;
}
```

Сначала должен быть создан объект базового класса, поэтому если опустить вызов конструктора базового класса, программа воспользуется конструктором базового класса по умолчанию.

Следовательно, предыдущий код аналогичен следующему:

```
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
    const string & ln, bool ht) // : TableTennisPlayer()
{
   rating = r;
}
```

Кроме случаев, когда точно нужно использовать конструктор по умолчанию, следует предусмотреть явный вызов соответствующего конструктора базового класса.

Теперь рассмотрим код для второго конструктора:

```
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
    : TableTennisPlayer(tp)
{
   rating = r;
}
```

Информация `TableTennisPlayer` также передается конструктору `TableTennis Player`:

```
TableTennisPlayer(tp)
```

Поскольку `tp` имеет тип `const TableTennisPlayer &`, при этом вызывается конструктор копирования базового класса. Конструктор копирования в базовом классе не определен, однако в главе 12 уже было сказано, что если конструктор копирования необходим, но не был определен, компилятор генерирует его автоматически. В данном случае вполне годится неявный конструктор, который выполняет почленное копирование, т.к. класс не использует непосредственно динамическое выделение памяти. (Члены `string` используют динамическое выделение памяти, однако вспомните, что при почленном копировании таких членов применяется конструктор копирования класса `string`.)

При необходимости для членов производного класса можно также использовать список инициализаторов. В этом случае в списке вместо имени класса используется имя члена. Таким образом, второй конструктор можно записать и в следующем виде:

```
// Альтернативный вариант
RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
   : TableTennisPlayer(tp), rating(r)
{
}
```

Ниже перечислены основные моменты, которые следует знать о конструкторах для производных классов.

- Сначала создается объект базового класса.
- Конструктор производного класса должен передавать информацию базового класса конструктору базового класса через список инициализаторов членов.
- Конструктор производного класса должен инициализировать данные-члены, добавленные в производном классе.

В настоящем примере не предусмотрены явные деструкторы, поэтому используются неявные деструкторы. Уничтожение объектов происходит в порядке, обратном порядку их создания. То есть сначала выполняется тело деструктора производного класса, а затем автоматически вызывается деструктор базового класса.

На заметку!

При создании объекта производного класса программа сначала вызывает конструктор базового класса, а потом конструктор производного класса. Конструктор базового класса отвечает за инициализацию унаследованных данных-членов. Конструктор производного класса отвечает за инициализацию всех добавленных данных-членов. Конструктор производного класса всегда вызывает конструктор базового класса. Чтобы указать, *какой* конструктор базового класса следует использовать, можно задействовать синтаксис списка инициализаторов. В противном случае вызывается конструктор базового класса по умолчанию.

Когда объект производного класса прекращает свое существование, программа сначала вызывает деструктор производного класса, а затем деструктор базового класса.

Списки инициализаторов членов

Конструктор для производного класса может использовать механизм списка инициализаторов для передачи значений конструктору базового класса. Вот пример:

```
derived::derived(тип1 x, тип2 y) : base(x,y)  // список инициализаторов
{
   ...
}
```

Здесь `derived` — производный класс, `base` — базовый класс, а `x` и `y` — переменные, которые используются конструктором базового класса. Если, к примеру, конструктор производного класса получает аргументы `10` и `12`, то данный механизм передает значения `10` и `12` конструктору базового класса, который принимает аргументы указанных типов. За исключением случая виртуальных базовых классов (см. главу 14), класс может передавать значения только своему непосредственному базовому классу. Однако принимающий класс может использовать тот же механизм для передачи информации своему непосредственному базовому классу и т.д. Если в списке инициализаторов членов не предусмотрен конструктор базового класса, программа использует конструктор базового класса по умолчанию. Список инициализаторов членов может использоваться *только* в конструкторах.

Использование производного класса

Чтобы использовать производный класс, программе необходимо иметь доступ к объявлениям базового класса. В листинге 13.4 оба объявления классов располагаются в одном и том же заголовочном файле. Каждому классу можно предоставить собственный заголовочный файл, однако поскольку классы зависят друг от друга, гораздо удобнее хранить объявления классов вместе.

Листинг 13.4. `tabtenn1.h`

```
// tabtenn1.h -- базовый класс для клуба по настольному теннису
#ifndef TABTENN1_H_
#define TABTENN1_H_
#include <string>
using std::string

// Простой базовый класс
class TableTennisPlayer
{
private:
   string firstname;
   string lastname
   bool hasTable;
```

```
public:
   TableTennisPlayer (const string & fn = "none",
                      const string & ln = "none", bool ht=false);
   void Name() const;
   bool HasTable() const { return hasTable; } ;
   void ResetTable(bool v) { hasTable = v; };
};

// Простой производный класс
class RatedPlayer : public TableTennisPlayer
{
private:
   unsigned int rating;
public:
   RatedPlayer (unsigned int r = 0, const string & fn = "none",
                const string & ln = "none", bool ht = false);
   RatedPlayer(unsigned int r, const TableTennisPlayer & tp);
   unsigned int Rating() const { return rating; }
   void ResetRating (unsigned int r) {rating = r;}
};

#endif
```

В листинге 13.5 представлены определения методов для обоих классов. Как уже было сказано, можно использовать отдельные файлы, однако проще хранить определения вместе.

Листинг 13.5. `tabtenn1.cpp`

```
// tabtenn1.cpp -- методы простого базового класса
#include "tabtenn1.h"
#include <iostream>

TableTennisPlayer::TableTennisPlayer (const string & fn,
   const string & ln, bool ht) : firstname(fn),
   lastname(ln), hasTable(ht) {}

void TableTennisPlayer::Name() const
{
   std::cout << lastname << ", " << firstname;
}

// Методы RatedPlayer
RatedPlayer::RatedPlayer(unsigned int r, const string & fn,
   const string & ln, bool ht) : TableTennisPlayer(fn, ln, ht)
{
   rating = r;
}

RatedPlayer::RatedPlayer(unsigned int r, const TableTennisPlayer & tp)
   : TableTennisPlayer(tp), rating(r)
{
}
```

Код в листинге 13.6 создает объекты как класса `TableTennisPlayer`, так и класса `RatedPlayer`. Обратите внимание, что объекты обоих классов могут использовать методы `Name()` и `HasTable()` класса `TableTennisPlayer`.

Листинг 13.6. usett1.cpp

```
// usett1.cpp -- использование базового и производного классов
#include <iostream>
#include "tabtenn1.h"
int main ( void )
{
   using std::cout;
   using std::endl;
   TableTennisPlayer player1("Tara", "Boomdea", false);
   RatedPlayer rplayer1(1140, "Mallory", "Duck", true);

   rplayer1.Name();  // объект производного класса использует метод базового класса
   if (rplayer1.HasTable())
      cout << ": has a table.\n";
   else
      cout << ": hasn't a table.\n";

   player1.Name();   // объект базового класса использует метод базового класса
   if (player1.HasTable())
      cout << ": has a table";
   else
      cout << ": hasn't a table.\n";

   cout << "Name: ";
   rplayer1.Name();
   cout << "; Rating: " << rplayer1.Rating() << endl;

   // Инициализация объекта RatedPlayer с помощью объекта TableTennisPlayer
   RatedPlayer rplayer2(1212, player1);
   cout << "Name: ";
   rplayer2.Name();
   cout << "; Rating: " << rplayer2.Rating() << endl;
   return 0;
}
```

Ниже показан вывод программы, представленной в листингах 13.4, 13.5 и 13.6:

```
Duck, Mallory: has a table.
Boomdea, Tara: hasn't a table.
Name: Duck, Mallory; Rating: 1140
Name: Boomdea, Tara; Rating: 1212
```

Особые отношения между производным и базовым классами

Производный класс имеет особые отношения с базовым классом. Одно из них вы уже видели — объект производного класса может использовать методы базового класса, если эти методы не закрытые:

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
rplayer1.Name();  //объект производного класса использует метод базового класса
```

Еще два важных отношения: указатель базового класса может указывать на объект производного класса без явного привсдения типа, а ссылка базового класса может ссылаться на объект производного класса без явного приведения типа:

```
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
TableTennisPlayer & rt = rplayer;
TableTennisPlayer * pt = &rplayer;
rt.Name();        // вызов Name() с помощью ссылки
pt->Name();       // вызов Name() с помощью указателя
```

Однако указатель или ссылка на базовый класс позволяет вызывать методы только базового класса, поэтому с помощью rt или pt невозможно обратиться, например, к методу ResetRanking().

Обычно C++ требует, чтобы ссылочные типы и типы указателей соответствовали присваиваемым типам, но для наследования это правило ослаблено. Правда, оно ослаблено только в одном направлении. Ссылкам и указателям производного класса запрещено присваивать объекты и адреса базового класса:

```
TableTennisPlayer player("Betsy", "Bloop", true);
RatedPlayer & rr = player;   // НЕ РАЗРЕШЕНО
RatedPlayer * pr = player;   // НЕ РАЗРЕШЕНО
```

Оба эти набора правил имеют смысл. Например, рассмотрим возможные последствия того, что ссылка базового класса будет указывать на производный объект. В таком случае ссылку базового класса можно использовать для вызова методов базового класса для объекта производного класса. Поскольку производный класс наследует методы и данные-члены базового класса, это не вызывает проблем. А теперь представим, что может произойти, если будет возможно присвоить объект базового класса ссылке на производный класс. Ссылка на производный класс должна иметь возможность вызывать методы производного класса для базового объекта, а это может привести к проблемам. Например, использование метода RatedPlayer::Rating() для объекта TableTennisPlayer не имеет смысла, поскольку в объекте TableTennisPlayer нет члена rating.

Тот факт, что ссылки и указатели базового класса могут ссылаться на объекты производного класса, имеет несколько интересных следствий. Одно из них — функции, определенные со ссылкой или указателем на базовый класс в качестве аргументов, могут использоваться с объектами как базового, так и производного класса. Для примера рассмотрим следующую функцию:

```
void Show(const TableTennisPlayer & rt)
{
   using std::cout;
   cout << "Name: ";
   rt.Name();
   cout << "\nTable: ";
   if (rt.HasTable())
      cout << "yes\n";
   else
      cout << "no\n";
}
```

Формальный параметр rt является ссылкой на базовый класс, следовательно, он может указывать на объект базового или производного класса. Поэтому метод Show() можно использовать и с аргументом TableTennis, и с аргументом RatedPlayer:

```
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Show(player1);    // работает с аргументом TableTennisPlayer
Show(rplayer1);   // работает с аргументом RatedPlayer
```

Аналогичное отношение справедливо для функции, у которой формальный параметр представляет собой указатель на базовый класс. Ее можно вызвать как с адресом объекта базового класса, так и с адресом объекта производного класса в качестве фактического аргумента:

```
void Wohs(const TableTennisPlayer * pt); // функция с параметром-указателем
...
TableTennisPlayer player1("Tara", "Boomdea", false);
RatedPlayer rplayer1(1140, "Mallory", "Duck", true);
Wohs(&player1);   // работает с аргументом TableTennisPlayer *
Wohs(&rplayer1);  // работает с аргументом RatedPlayer *
```

Свойство совместимости ссылок также позволяет инициализировать объект базового класса значением объекта производного класса, хоть и не напрямую. Пусть имеется следующий код:

```
RatedPlayer olaf1(1840, "Olaf", "Loaf", true);
TableTennisPlayer olaf2(olaf1);
```

Точным соответствием для инициализации `olaf2` был бы конструктор со следующим прототипом:

```
TableTennisPlayer(const RatedPlayer &);   // не существует
```

В определениях класса нет такого конструктора, однако существует неявный конструктор копирования:

```
// Неявный конструктор копирования
TableTennisPlayer(const TableTennisPlayer &);
```

Формальный параметр является ссылкой на базовый тип, значит, он может указывать и на производный тип. Поэтому при попытке инициализировать `olaf2` значением `olaf1` используется данный конструктор, который копирует члены `firstname`, `lastname` и `hasTable`. То есть он инициализирует `olaf2` значением объекта `TableTennisPlayer`, вложенного в объект `olaf1` типа `RatedPlayer`.

Аналогично объекту базового класса можно присвоить объект производного класса:

```
RatedPlayer olaf1(1840, "Olaf", "Loaf", true);
TableTennisPlayer winner;
winner = olaf1;   // присваивание производного объекта базовому объекту
```

В этом случае программа использует неявную перегруженную операцию присваивания:

```
TableTennisPlayer & operator=(const TableTennisPlayer &) const;
```

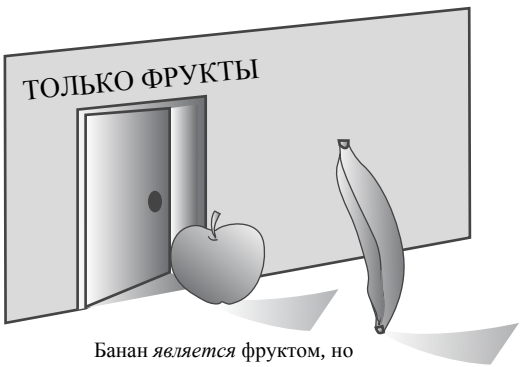
Ссылка базового класса указывает на объект производного класса, и в `winner` копируется только часть `olaf1`, соответствующая базовому классу.

Наследование: отношение *является*

Особое отношение между производным и базовым классами основано на внутренней модели наследования C++. В действительности в C++ имеется три варианта наследования: открытое, защищенное и закрытое. Открытое наследование является наиболее общей формой, и оно моделирует отношение *является* (is-a). Это условное обозначение того, что объект производного класса должен также быть объектом базового класса. Все, что можно делать с объектом базового класса, должно быть возможным и для объекта производного класса. Предположим, например, что у нас есть класс `Fruit`, представляющий фрукт. Он содержит, скажем, вес и количество калорий. Поскольку банан — разновидность фрукта, от класса `Fruit` можно породить класс `Banana`. Новый класс унаследует все данные-члены исходного класса, то есть объект `Banana` будет иметь члены, представляющие вес и содержание калорий в банане. Новый класс `Banana` может также иметь дополнительные члены, характерные для бананов, но не для фруктов вообще — например, индекс кожуры "института бананов".

Поскольку производный класс может содержать дополнительные свойства, то, на верно, та кое отношение было бы точнее назвать "является разновидностью", однако общепринятым стал термин *является*.

Чтобы лучше понять отношение *является*, рассмотрим несколько примеров, которые не соответствуют данной модели. Открытое наследование не моделирует отношение *содержит* (has-a). Обед, к примеру, может содержать фрукт, однако обед не обязательно является фруктом. Следовательно, не стоит порождать класс `Lunch` от класса `Fruit`, пытаясь поместить фрукт в обед. Правильным способом добавления фрукта в обед является отношение *содержит*: обед содержит фрукт. Как будет показано в главе 14, это легче всего моделировать, включив объект `Fruit` в качестве члена данных класса `Lunch` (рис. 13.3).



Банан *является* фруктом, но обед *содержит* банан.

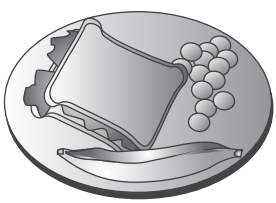


***Рис. 13.3.** Отношения является и содержит*

Открытое наследование не моделирует отношение *подобен* (is-like-a) – т.е. оно не делает сравнений. Очень часто говорят, что адвокаты похожи на акул. Но это не означает, что адвокат на самом деле акула. Например, акулы могут жить под водой. Поэтому не следует порождать класс `Lawyer` от класса `Shark`. Наследование может добавлять свойства в базовый класс, но оно не удаляет свойства из базового класса. В некоторых случаях для работы с общими характеристиками можно создать класс, содержащий эти характеристики, а затем использовать данный класс либо в отношении *является*, либо в отношении *содержит* для определения взаимосвязанных классов.

Открытое наследование не моделирует отношение *реализован как* (is-implemented-as-a). Например, можно реализовать стек с помощью массива. Однако будет неправильно породить класс `Stack` от класса `Array`. Стек – это не массив; например, индексация массива не является свойством стека. Стек можно реализовать и другим способом, допустим, с помощью связного списка. Правильнее будет скрыть реализацию посредством массива, введя в класс стека закрытый объект-член `Array`.

Открытое наследование не моделирует отношение *использует* (uses-a). Например, компьютер может использовать лазерный принтер, но не имеет смысла порождать

класс Printer от класса Computer, и наоборот. Однако можно разработать дружественные функции или классы для управления обменом данными между объектами Printer и Computer.

В C++ ничто не мешает использовать открытое наследование для моделирования отношений *содержит*, *реализован как* или *использует*. Однако это, как правило, приводит к проблемам при программировании. Поэтому мы будем придерживаться отношений *является*.

Полиморфное открытое наследование

Пример наследования RatedPlayer является достаточно простым. Объекты производного класса используют методы базового класса без изменений. Однако возможны ситуации, когда метод должен обладать разным поведением в производном и базовом классах. Другими словами, поведение конкретного метода может отличаться в зависимости от объекта, который его вызывает. Такое более сложное поведение называется *полиморфным*, поскольку у метода имеется несколько моделей поведения в зависимости от контекста. Существуют два основных механизма для реализации полиморфного открытого наследования:

- переопределение методов базового класса в производном классе;
- использование виртуальных методов.

Рассмотрим еще один пример. Вы получили неплохой опыт во время работы в клубе "Webtown Social Club", и теперь готовы стать ведущим программистом банка "Pontoon National Bank". Первое задание, которое дают вам в банке – разработка двух классов. Один класс представляет чековый счет Brass Account, а второй – чековый счет Brass Plus, в котором добавлено свойство защиты от овердрафта (превышения кредита). То есть если клиент выписывает чек на сумму, которая больше (но не намного), чем его баланс, то банк оплачивает этот чек, предоставляя клиенту кредит для дополнительного платежа, и начисляет процент на этот кредит. Эти два вида счета можно охарактеризовать через данные, которые требуют хранения, и допустимые операции.

Для начала, вот информация для счета Brass Account:

- имя клиента;
- номер счета;
- текущий баланс.

А вот необходимые операции:

- создание счета;
- внесение денег на счет;
- снятие денег со счета;
- вывод состояния счета.

Счет Brass Plus должен содержать все свойства Brass Account, а также следующие дополнительные информационные элементы:

- максимальное значение овердрафта;
- процентная ставка, начисляемая на овердрафт;
- величина овердрафта, которую клиент должен банку на данный момент.

Дополнительные операции не нужны, однако две операции необходимо реализовать по-другому:

- операция снятия денег для счета Brass Plus должна содержать защиту от овердрафта;
- операция вывода должна отображать всю дополнительную информацию, необходимую для счета Brass Plus.

Предположим, что вы назвали один класс `Brass`, а второй — `BrassPlus`. Нужно ли порождать `BrassPlus` от `Brass` с помощью открытого наследования? Чтобы ответить на этот вопрос, сначала ответьте на другой: соответствует ли класс `BrassPlus` критерию отношения *является*? Конечно. Все, что верно для объектов `Brass`, будет верно и для объектов `BrassPlus`. Оба они хранят имя клиента, номер счета и баланс. Оба счета позволяют вносить и снимать деньги, а также выводить информацию о текущем балансе. Учтите, что отношение *является* в общем случае не симметрично. Фрукт не обязательно является бананом; аналогично, объект `Brass` не обладает всеми возможностями объекта `BrassPlus`.

Разработка классов `Brass` и `BrassPlus`

Структура класса для счета Brass Account не вызывает трудностей, однако банк не предоставил достаточную информацию о том, как работает учет овердрафта. В ответ на ваш запрос о дополнительных деталях дружелюбный работник банка "Pontoon National Bank" сообщил следующее.

- Счет Brass Plus ограничивает сумму денег, которую банк может одолжить клиенту для покрытия овердрафта. Значение по умолчанию — $500, но для некоторых клиентов может быть установлен другой лимит.
- Банк может изменять предел овердрафта для клиента.
- Счет Brass Plus предусматривает начисление процентов на ссуду. Значение по умолчанию — 11,125%, но для некоторых клиентов может быть установлена другая ставка.
- Банк может изменять процентную ставку клиента.
- Счет учитывает, какую сумму клиент должен банку (ссуда овердрафта плюс проценты). Клиент не может погасить эту сумму через обычный вклад или переводом денег с другого счета. Он должен заплатить наличными специальному банковскому служащему, который, если понадобится, будет разыскивать клиента. Как только долг погашен, на счету указывается нулевое значение задолженности.

Последнее свойство не в банковских традициях, однако, к счастью, оно упрощает программирование.

Приведенный перечень наталкивает на мысль, что для нового класса нужны конструкторы, которые предоставляют информацию о состоянии счета и включают в себя предельное значение долга со значением по умолчанию $500 и процентную ставку со значением по умолчанию 11,125%. Должны также существовать методы для изменения предельной суммы долга, процентной ставки и текущего долга. Это все, что требуется добавить в класс `Brass`. И это будет сделано в объявлении класса `BrassPlus`.

Информация о данных двух классах позволяет построить объявления классов, которые похожи на приведенные в листинге 13.7.

Листинг 13.7. brass.h

```
// brass.h -- классы банковских счетов
#ifndef BRASS_H_
#define BRASS_H_
#include <string>

// Класс счета Brass Account
class Brass
{
private:
   std::string fullName;
   long acctNum;
   double balance;
public:
   Brass(const std::string & s = "Nullbody", long an = -1,
         double bal = 0.0);
   void Deposit(double amt);
   virtual void Withdraw(double amt);
   double Balance() const;
   virtual void ViewAcct() const;
   virtual ~Brass() {}
};

// Класс счета Brass Plus
class BrassPlus : public Brass
{
private:
   double maxLoan;
   double rate;
   double owesBank;
public:
   BrassPlus(const std::string & s = "Nullbody", long an = -1,
             double bal = 0.0, double ml = 500,
             double r = 0.11125);
   BrassPlus(const Brass & ba, double ml = 500, double r = 0.11125);
   virtual void ViewAcct()const;
   virtual void Withdraw(double amt);
   void ResetMax(double m) { maxLoan = m; }
   void ResetRate(double r) { rate = r; };
   void ResetOwes() { owesBank = 0; }
};

#endif
```

В листинге 13.7 следует обратить внимание на ряд перечисленных ниже моментов.

- Класс `BrassPlus` добавляет в класс `Brass` три новых закрытых члена данных и три новых открытых функции-члена.
- Оба класса объявляют методы `ViewAcct()` и `Withdraw()`; однако вести себя они будут в разных объектах по-разному.
- При объявлении `ViewAcct()` и `Withdraw()` в классе `Brass` использовано новое ключевое слово `virtual`. Эти методы теперь называются *виртуальными*.
- Класс `Brass` также объявляет виртуальный деструктор, который ничего не делает.

Первый пункт в данном списке для нас не нов. Класс RatedPlayer делал нечто подобное, когда добавлял в класс TableTennisPlayer новый член данных и два новых метода.

Вторым важным моментом в списке является то, как в объявлениях задается, что методы в производном классе ведут себя по-другому. Два прототипа ViewAcct() указывают, что должны существовать два отдельных определения метода. Уточненным именем для версии базового класса служит Brass::ViewAcct(), а для производного класса — BrassPlus::ViewAcct(). Для определения нужной версии программа будет использовать тип объекта:

```
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
dom.ViewAcct();      // вызывается Brass::ViewAcct()
dot.ViewAcct();      // вызывается BrassPlus::ViewAcct()
```

Аналогично существуют и две версии Withdraw(): одна для объектов Brass и одна — для объектов BrassPlus. Методы, которые ведут себя одинаково для обоих классов, такие как Deposit() и Balance(), объявлены только в базовом классе.

Третий вопрос (применение virtual) сложнее, чем первые два. Он определяет, какой метод используется, если метод вызывается не объектом, а ссылкой или указателем. Без ключевого слова virtual программа выбирает метод, основываясь на типе ссылки или указателя. Но если присутствует ключевое слово virtual, программа выбирает метод, основываясь на типе объекта, на который указывает ссылка или указатель. Вот как ведет себя программа, если функция ViewAcct() не является виртуальной:

```
// Поведение не виртуальной функции ViewAcct()
// Метод выбирается в соответствии с типом ссылки
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
Brass & b1_ref = dom;
Brass & b2_ref = dot;
b1_ref.ViewAcct();   // вызывается Brass::ViewAcct()
b2_ref.ViewAcct();   // вызывается Brass::ViewAcct()
```

Ссылочные переменные относятся к типу Brass, поэтому выбирается Brass::ViewAccount(). Использование указателей на Brass вместо ссылки дает аналогичное поведение.

Для сравнения продемонстрируем поведение при виртуальной функции ViewAcct():

```
// Поведение виртуальной функции ViewAcct()
// Метод выбирается в соответствии с типом объекта
Brass dom("Dominic Banker", 11224, 4183.45);
BrassPlus dot("Dorothy Banker", 12118, 2592.00);
Brass & b1_ref = dom;
Brass & b2_ref = dot;
b1_ref.ViewAcct();   // вызывается Brass::ViewAcct()
b2_ref.ViewAcct();   // вызывается BrassPlus::ViewAcct()
```

В этом случае обе ссылки относятся к типу Brass, но b2_ref ссылается на объект BrassPlus, поэтому для него вызывается BrassPlus::ViewAcct(). Использование указателей на Brass вместо ссылки обеспечивает аналогичное поведение.

Оказывается, как будет показано ниже, такое поведение виртуальных функций весьма удобно. Поэтому общей рекомендацией будет объявление в базовом классе в качестве виртуальных тех методов, которые могут быть переопределены в производ-

ном классе. Если метод объявлен в базовом классе как виртуальный, он автоматически является виртуальным и в производном классе. Однако в объявлениях производного класса также рекомендуется указывать, какие функции являются виртуальными, с помощью ключевого слова `virtual`.

Четвертый момент заключается в том, что в базовом классе объявлен виртуальный деструктор. Это необходимо для правильной последовательности вызовов деструкторов при уничтожении производного объекта. Данный вопрос мы обсудим более подробно ниже в данной главе.

> **На заметку!**
>
> Если планируется переопределять какой-либо метод базового класса в производном классе, то обычно такой метод объявляется в базовом классе как виртуальный. Тогда программа выбирает версию метода, основываясь на типе объекта, а не на типе ссылки и указателя. Также в базовом классе принято объявлять виртуальный деструктор.

Реализации классов

Следующий шаг — подготовка реализации классов. Часть этой работы уже была сделана с помощью встроенных определений функций в заголовочном файле.

Листинг 13.8 содержит остальные определения методов. Обратите внимание, что ключевое слово `virtual` присутствует только в прототипах методов в объявлении класса, но не в определениях методов в листинге 13.8.

Листинг 13.8. `brass.cpp`

```
// brass.cpp -- методы классов банковских счетов
#include <iostream>
#include "brass.h"
using std::cout;
using std::endl;
using std::string;

// Для целей форматирования
typedef std::ios_base::fmtflags format;
typedef std::streamsize precis;
format setFormat();
void restore(format f, precis p);

// Методы Brass
Brass::Brass(const string & s, long an, double bal)
{
    fullName = s;
    acctNum = an;
    balance = bal;
}
void Brass::Deposit(double amt)
{
    if (amt < 0)
        cout << "Negative deposit not allowed; "
             << "deposit is cancelled.\n";  // отрицательный вклад не допускается
    else
        balance += amt;
}
void Brass::Withdraw(double amt)
{
    // Установка формата ###.##
    format initialState = setFormat();
```

```
    precis prec = cout.precision(2);
    if (amt < 0)
        cout << "Withdrawal amount must be positive; "
             << "withdrawal canceled.\n";  // снимаемая сумма должна быть положительной
    else if (amt <= balance)
        balance -= amt;
    else
        cout << "Withdrawal amount of $" << amt
             << " exceeds your balance.\n"
             << "Withdrawal canceled.\n";  // снимаемая сумма превышает текущий баланс
    restore(initialState, prec);
}

double Brass::Balance() const
{
    return balance;
}

void Brass::ViewAcct() const
{
    // Установка формата ###.##
    format initialState = setFormat();
    precis prec = cout.precision(2);
    cout << "Client: " << fullName << endl;              // клиент
    cout << "Account Number: " << acctNum << endl;       // номер счета
    cout << "Balance: $" << balance << endl;             // баланс
    restore(initialState, prec);         // восстановление исходного формата
}

// Методы BrassPlus
BrassPlus::BrassPlus(const string & s, long an, double bal,
                 double ml, double r) : Brass(s, an, bal)
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
        : Brass(ba)          // используется неявный конструктор копирования
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}

// Переопределение реализации метода ViewAcct()
void BrassPlus::ViewAcct() const
{
    // Установка формата ###.##
    format initialState = setFormat();
    precis prec = cout.precision(2);
    Brass::ViewAcct();                                  // отображение базовой части
    cout << "Maximum loan: $" << maxLoan << endl;       // максимальный заем
    cout << "Owed to bank: $" << owesBank << endl;      // долг банку
    cout.precision(3);                                  // формат ###.###
    cout << "Loan Rate: " << 100 * rate << "%\n";       // процент на заем
    restore(initialState, prec);
}
```

```
// Переопределение реализации метода Withdraw()
void BrassPlus::Withdraw(double amt)
{
   // Установка формата ###.##
   format initialState = setFormat();
   precis prec = cout.precision(2);
   double bal = Balance();
   if (amt <= bal)
     Brass::Withdraw(amt);
   else if ( amt <= bal + maxLoan - owesBank)
   {
     double advance = amt - bal;
     owesBank += advance * (1.0 + rate);
     cout << "Bank advance: $" << advance << endl;                // аванс банка
     cout << "Finance charge: $" << advance * rate << endl;       // долг банку
     Deposit(advance);
     Brass::Withdraw(amt);
   }
   else
     cout << "Credit limit exceeded. Transaction cancelled.\n"; //предел кредита превышен
   restore(initialState, prec);
}
format setFormat()
{
   // Установка формата ###.##
   return cout.setf(std::ios_base::fixed,
   std::ios_base::floatfield);
}
void restore(format f, precis p)
{
   cout.setf(f, std::ios_base::floatfield);
   cout.precision(p);
}
```

Прежде чем приступить к изучению деталей листинга 13.8, таких как управление форматированием в некоторых методах, рассмотрим те аспекты, которые относятся непосредственно к наследованию. Вспомните, что производный класс не имеет прямого доступа к закрытым данным базового класса, и для доступа к этим данным ему приходится использовать открытые методы базового класса. Средства доступа зависят от метода. Конструкторы применяют одни способы, а остальные функции-члены — другие.

Для инициализации закрытых данных базового класса конструкторы производного класса используют списки инициализаторов членов. Этот прием применяется как в конструкторах класса `RatedPlayer`, так и в конструкторах `BrassPlus`:

```
BrassPlus::BrassPlus(const char & s, long an, double bal,
         double ml, double r) : Brass(s, an, bal)
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}
BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
        : Brass(ba)   // используется неявный конструктор копирования
{
    maxLoan = ml;
    owesBank = 0.0;
    rate = r;
}
```

Каждый из этих конструкторов использует список инициализаторов членов для передачи информации базового класса конструктору базового класса, а затем — тело конструктора для инициализации новых элементов данных, добавляемых классом BrassPlus.

Методы не могут применять синтаксис списка инициализаторов членов, если это не конструкторы. Однако метод производного класса может вызвать открытый метод базового класса. Например, не считая аспекта форматирования, основной код функции ViewAcct() в версии BrassPlus выглядит так:

```
// Переопределение реализации метода ViewAcct()
void BrassPlus::ViewAcct() const
{
    ...
    Brass::ViewAcct();                              // отображение базовой части
    cout << "Maximum loan: $" << maxLoan << endl;  // максимальный заем
    cout << "Owed to bank: $" << owesBank << endl; // долг банку
    cout.precision(3);                             // формат ###.###
    cout << "Loan Rate: " << 100 * rate << "%\n";  // процент на заем
    ...
}
```

То есть BrassPlus::ViewAcct() выводит добавленные данные-члены BrassPlus и вызывает метод базового класса Brass::ViewAcct() для вывода данных-членов базового класса. Использование операции разрешения контекста в методе производного класса для вызова метода базового класса — стандартный прием.

Очень важно то, что в коде применяется операция разрешения контекста. Предположим, что вместо предыдущего кода написан такой:

```
// Переопределение реализации метода ViewAcct()
void BrassPlus::ViewAcct() const
{
    ...
    ViewAcct(); // рекурсивный вызов
    ...
}
```

При отсутствии операции разрешения контекста компилятор считает, что ViewAcct() — это BrassPlus::ViewAcct(), и создает рекурсивную функцию без завершения — что совсем не хорошо.

Теперь рассмотрим метод BrassPlus::Withdraw(). Если клиент снимает сумму, превышающую баланс, то метод должен оформить ссуду. Он может применить Brass::Withdraw() для доступа к члену баланса, но Brass::Withdraw() выдает сообщение об ошибке, если снимаемая сумма превышает баланс. В данной реализации можно избежать этого сообщения, если воспользоваться методом Deposit() для открытия ссуды, а затем, при наличии достаточных средств, вызвать Brass::Withdraw():

```
// Переопределение реализации метода Withdraw()
void BrassPlus::Withdraw(double amt)
{
    ...
    double bal = Balance();
    if (amt <= bal)
        Brass::Withdraw(amt);
    else if ( amt <= bal + maxLoan - owesBank)
    {
        double advance = amt - bal;
```

```
        owesBank += advance * (1.0 + rate);
        cout << "Bank advance: $" << advance << endl;            // аванс банка
        cout << "Finance charge: $" << advance * rate << endl;   // долг банку
        Deposit(advance);
        Brass::Withdraw(amt);
    }
    else
        cout << "Credit limit exceeded. Transaction cancelled.\n";
              // предел кредита превышен
    ...
}
```

Обратите внимание, что для определения исходного баланса метод использует функцию базового класса `Balance()`. Код не обязан применять разрешение контекста для `Balance()`, поскольку этот метод не переопределялся в производном классе.

Методы `ViewAcct()` и `Withdraw()` применяют методы форматирования `setf()` и `precision()` для вывода величин с плавающей запятой в виде с фиксированной точкой и с двумя знаками после десятичной точки. После установки этих режимов они так и остаются, поэтому методы возвращают режим форматирования в состояние, которое было до их вызова. В листингах 8.8 и 10.5 используются похожие подходы. Чтобы не дублировать код, часть действий по форматированию вынесена во вспомогательные функции:

```
// Для целей форматирования
typedef std::ios_base::fmtflags format;
typedef std::streamsize precis;
format setFormat();
void restore(format f, precis p);
```

Функция `setFormat()` устанавливает формат с фиксированной точкой и возвращает предыдущие настройки:

```
format setFormat()
{
  // Установка формата
  return cout.setf(std::ios_base::fixed,
        std::ios_base::floatfield);
}
```

А функция `restore()` восстанавливает формат и точность:

```
void restore(format f, precis p)
{
  cout.setf(f, std::ios_base::floatfield);
  cout.precision(p);
}
```

За дополнительными сведениями о форматировании обращайтесь в главу 17.

Использование классов `Brass` *и* `BrassPlus`

В листинге 13.9 представлен код, тестирующий классы `Brass` и `BrassPlus`.

Листинг 13.9. `usebrass1.cpp`

```
// usebrass1.cpp -- тестирование классов банковских счетов
// Компилировать вместе с brass.cpp
#include <iostream>
#include "brass.h"
```

```
int main()
{
   using std::cout;
   using std::endl;
   Brass Piggy("Porcelot Pigg", 381299, 4000.00);
   BrassPlus Hoggy("Horatio Hogg", 382288, 3000.00);
   Piggy.ViewAcct();
   cout << endl;
   Hoggy.ViewAcct();
   cout << endl;
   cout << "Depositing $1000 into the Hogg Account:\n";
   Hoggy.Deposit(1000.00);
   cout << "New balance: $" << Hoggy.Balance() << endl;
   cout << "Withdrawing $4200 from the Pigg Account:\n";
   Piggy.Withdraw(4200.00);
   cout << "Pigg account balance: $" << Piggy.Balance() << endl;
   cout << "Withdrawing $4200 from the Hogg Account:\n";
   Hoggy.Withdraw(4200.00);
   Hoggy.ViewAcct();
   return 0;
}
```

Ниже показан вывод программы из листинга 13.9:

```
Client: Porcelot Pigg
Account Number: 381299
Balance: $4000.00

Client: Horatio Hogg
Account Number: 382288
Balance: $3000.00
Maximum loan: $500.00
Owed to bank: $0.00
Loan Rate: 11.125%

Depositing $1000 into the Hogg Account:
New balance: $4000
Withdrawing $4200 from the Pigg Account:
Withdrawal amount of $4200.00 exceeds your balance.
Withdrawal canceled.
Pigg account balance: $4000
Withdrawing $4200 from the Hogg Account:
Bank advance: $200.00
Finance charge: $22.25
Client: Horatio Hogg
Account Number: 382288
Balance: $0.00
Maximum loan: $500.00
Owed to bank: $222.25
Loan Rate: 11.125%
```

Демонстрация поведения виртуальных методов

В листинге 13.9 методы вызываются объектами, а не указателями или ссылками, поэтому программа не использует возможности виртуальных методов. Давайте рассмотрим пример, в котором задействованы виртуальные методы. Предположим, что вам требуется управлять смесью счетов `Brass` и `BrassPlus`. Было бы удобно иметь единственный массив, хранящий набор объектов `Brass` и `BrassPlus`, но это невозможно: каждый элемент массива должен относиться к одному и тому же типу, а

Brass и BrassPlus — два различных типа. Однако можно создать массив указателей на Brass. В этом случае все элементы будут одного типа, но благодаря модели общедоступного наследования указатель на Brass может указывать либо на объект Brass, либо на объект BrassPlus. То есть, по сути, у нас имеется способ представления коллекции данных более чем одного типа в едином массиве. Это и есть полиморфизм; в листинге 13.10 показан простой пример.

Листинг 13.10. usebrass2.cpp

```
// usebrass2.cpp -- пример полиморфизма
// Компилировать вместе с brass.cpp
#include <iostream>
#include <string>
#include "brass.h"
const int CLIENTS = 4;
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   Brass * p_clients[CLIENTS];
   std::string temp;
   long tempnum;
   double tempbal;
   char kind;
   for (int i = 0; i < CLIENTS; i++)
   {
      cout << "Enter client's name: ";                  // ввод имени клиента
      getline(cin,temp);
      cout << "Enter client's account number: ";        // ввод номера счета клиента
      cin >> tempnum;
      cout << "Enter opening balance: $";               // ввод начального баланса
      cin >> tempbal;
      cout << "Enter 1 for Brass Account or "
        << "2 for BrassPlus Account: "; // 1 -- Brass Account; 2 -- BrassPlus Account
      while (cin >> kind && (kind != '1' && kind != '2'))
         cout <<"Enter either 1 or 2: ";
      if (kind == '1')
         p_clients[i] = new Brass(temp, tempnum, tempbal);
      else
      {
         double tmax, trate;
         cout << "Enter the overdraft limit: $";       // ввод предельного овердрафта
         cin >> tmax;
         cout << "Enter the interest rate "
              << "as a decimal fraction: ";            // ввод процентной ставки
         cin >> trate;
         p_clients[i] = new BrassPlus(temp, tempnum, tempbal, tmax, trate);
      }
      while (cin.get() != '\n')
         continue;
   }
   cout << endl;
   for (int i = 0; i < CLIENTS; i++)
   {
      p_clients[i]->ViewAcct();
      cout << endl;
   }
```

```
    for (int i = 0; i < CLIENTS; i++)
    {
        delete p_clients[i];  // освобождение памяти
    }
    cout << "Done.\n";
    return 0;
}
```

Программа в листинге 13.10 позволяет определять тип добавляемого счета, а затем с помощью операции `new` создает и инициализирует объект соответствующего типа. Как вы должны помнить, вызов `getline(cin,temp)` читает строку ввода из `cin` и сохраняет ее в объекте типа `string` по имени `temp`.

Ниже показан пример выполнения программы из листинга 13.10:

```
Enter client's name: Harry Fishsong
Enter client's account number: 112233
Enter opening balance: $1500
Enter 1 for Brass Account or 2 for BrassPlus Account: 1
Enter client's name: Dinah Otternoe
Enter client's account number: 121213
Enter opening balance: $1800
Enter 1 for Brass Account or 2 for BrassPlus Account: 2
Enter the overdraft limit: $350
Enter the interest rate as a decimal fraction: 0.12
Enter client's name: Brenda Birdherd
Enter client's account number: 212118
Enter opening balance: $5200
Enter 1 for Brass Account or 2 for BrassPlus Account: 2
Enter the overdraft limit: $800
Enter the interest rate as a decimal fraction: 0.10
Enter client's name: Tim Turtletop
Enter client's account number: 233255
Enter opening balance: $688
Enter 1 for Brass Account or 2 for BrassPlus Account: 1

Client: Harry Fishsong
Account Number: 112233
Balance: $1500.00

Client: Dinah Otternoe
Account Number: 121213
Balance: $1800.00
Maximum loan: $350.00
Owed to bank: $0.00
Loan Rate: 12.00%

Client: Brenda Birdherd
Account Number: 212118
Balance: $5200.00
Maximum loan: $800.00
Owed to bank: $0.00
Loan Rate: 10.00%

Client: Tim Turtletop
Account Number: 233255
Balance: $688.00

Done.
```

Полиморфизм обеспечивается с помощью следующего кода:

```
for (int i = 0; i < CLIENTS; i++)
{
  p_clients[i]->ViewAcct();
  cout << endl;
}
```

Если элемент массива указывает на объект `Brass`, то вызывается `Brass::ViewAcct()`, а если на объект `BrassPlus` — то `BrassPlus::ViewAcct()`. Если бы функция `Brass::ViewAcct()` была объявлена как виртуальная, то во всех случаях вызывался бы метод `Brass:ViewAcct()`.

Необходимость в виртуальных деструкторах

Код в листинге 13.10, где используется операция `delete` для освобождения объектов, память под которые выделена операцией `new`, демонстрирует, зачем в базовом классе нужен виртуальный деструктор, даже если необходимости в нем вроде бы нет. Если деструкторы не виртуальные, то вызывается только деструктор, соответствующий типу указателя. Для листинга 13.10 это означает, что всегда будет вызываться только деструктор `Brass`, даже если указатель указывает на объект `BrassPlus`. Но при наличии виртуальных деструкторов, если указатель указывает на объект `BrassPlus`, вызывается деструктор `BrassPlus`. А когда деструктор `BrassPlus` завершает свою работу, он автоматически вызывает конструктор базового класса. Таким образом, применение виртуальных деструкторов гарантирует вызов деструкторов в корректной последовательности. В листинге 13.10 такое правильное поведение не принципиально, поскольку деструкторы ничего не делают. Однако, если, например, `BrassPlus` имел бы деструктор, выполняющий какие-то действия, то деструктор `Brass` обязательно должен быть виртуальным, даже если он и ничего не делает.

Статическое и динамическое связывание

Какой блок исполняемого кода выполняется, когда программа вызывает функцию? На этот вопрос должен ответить компилятор. Интерпретация вызова функции в исходном коде в виде выполнения определенной части кода называется *связыванием* имени функции. В C эта задача не представляет сложности, т.к. каждое имя функции соответствует отдельной функции. В C++ все несколько сложнее из-за перегрузки функций. Компилятор должен учесть не только имя, но и аргументы функции, чтобы определить, какую функцию использовать. Но все же такой тип связывания компилятор C или C++ может выполнить во время компиляции. Связывание, выполняемое во время компиляции, называется *статическим* (или *ранним*) *связыванием*. Однако виртуальные функции еще более усложняют ситуацию. Как показано в листинге 13.10, решение о том, какую функцию использовать, не может быть принято во время компиляции, поскольку компилятор не знает, с объектом какого типа собирается работать пользователь. Поэтому компилятор должен генерировать код, который позволяет выбирать нужный виртуальный метод во время работы программы. Такой процесс называется *динамическим* (или *поздним*) *связыванием*.

Теперь, когда вы ознакомились с работой виртуальных методов, рассмотрим этот процесс более подробно. Начнем с того, как C++ поддерживает совместимость типов указателей и ссылок.

Совместимость типов указателей и ссылок

Динамическое связывание в C++ связано с методами, вызываемыми по указателям и ссылкам, и отчасти управляется процессом наследования. Один из способов, с помо-

щью которого открытое наследование моделирует отношение *является*, заключается в обработке указателей и ссылок на объекты. Обычно в C++ запрещено присваивать адрес одного типа указателю другого типа. Также не разрешается ссылке одного типа ссылаться на другой тип:

```
double x = 2.5;
int * pi = &x;  // недопустимое присваиванеи: несоответствие типов указателей
long & rl = x;  // недопустимое присваиванеи: несоответствие типов ссылок
```

Однако, как уже было сказано, ссылка или указатель на базовый класс может ссылаться на объект производного класса без явного приведения типа. Например, допустимы такие инициализации:

```
BrassPlus dilly ("Annie Dill", 493222, 2000);
Brass * pb = &dilly;     // нормально
Brass & rb = dilly;      // нормально
```

Преобразование ссылки или указателя на производный класс в ссылку или указатель на базовый класс называется *восходящим приведением*. Оно всегда разрешено для открытого наследования и не требует явного приведения типа. Это правило является частью выражения отношения *является*. Объект `BrassPlus` является объектом `Brass` в том смысле, что он наследует все данные-члены и функции-члены класса `Brass`. Поэтому все, что можно делать с объектом `Brass`, можно делать и с объектом `BrassPlus`. И значит, функция, разработанная для управления ссылкой на `Brass`, может без проблем делать то же самое и для объекта `BrassPlus`. Аналогичный принцип применим и при передаче указателя на объект в качестве аргумента функции. Восходящее приведение транзитивно: если от класса `BrassPlus` породить класс `BrassPlusPlus`, то указатель или ссылка на `Brass` сможет ссылаться на объект `Brass`, `BrassPlus` или `BrassPlusPlus`.

Обратный процесс, т.е. преобразование указателя или ссылки на базовый класс в указатель или ссылку на производный класс, называется *нисходящим приведением*, и оно не разрешено без явного приведения типа. Дело в том, что в общем случае отношение *является* не симметрично. Производный класс может добавить новые данные-члены, и функции-члены класса, которые используют эти данные-члены, могут быть неприменимы для базового класса. Например, предположим, что от класса `Employee` (работник) порожден класс `Singer` (певец): в нем добавлен член данных, представляющий вокальный диапазон певца, и метод `range()`, который сообщает его значение. Применение метода `range()` к объекту `Employee` в общем случае бессмысленно. Но если бы было допустимо неявное приведение, то можно было бы случайно занести адрес объекта `Employee` в указатель на `Singer` и применить указатель для вызова метода `range()` (рис. 13.4).

Восходящее приведение также выполняется для вызовов функций со ссылками или указателями на базовый класс в качестве параметров.

Рассмотрим следующий фрагмент кода, предполагая, что каждая функция вызывает виртуальный метод `ViewAcct()`:

```
void fr(Brass & rb);     // использует rb.ViewAcct()
void fp(Brass * pb);     // использует pb->ViewAcct()
void fv(Brass b);        // использует b.ViewAcct()
int main()
{
   Brass b("Billy Bee", 123432, 10000.0);
   BrassPlus bp("Betty Beep", 232313, 12345.0);
   fr(b);                // использует Brass::ViewAcct()
```

```
    fr(bp);             // использует BrassPlus::ViewAcct()
    fp(b);              // использует Brass::ViewAcct()
    fp(bp);             // использует BrassPlus::ViewAcct()
    fv(b);              // использует Brass::ViewAcct()
    fv(bp);             // использует Brass::ViewAcct()
    ...
}
```

Передача по значению приводит к передаче в функцию `fv()` только компонента `Brass` из объекта `BrassPlus`. Однако из-за неявного восходящего приведения, которое выполняется со ссылками и указателями, функции `fr()` и `fp()` используют `Brass::ViewAcct()` для объектов `Brass` и `BrassPlus::ViewAcct()` для объектов `BrassPlus`.

```
class Employee                  //работник
{
private:
    char name[40];
    ...
public:
    void show_name();
    ...
};
class Singer : public Employee  //певец
{
    ...
public:
    void range();
    ...
};
...
Employee veep;
Singer trala;
...
Employee * pe = &trala;
Singer * ps = (Singer *) &veep;
...
pe->show_name();
ps->range();
```

Восходящее приведение — допускается неявное приведение типа

Нисходящее приведение — требуется явное приведение типа

Восходящее приведение безопасно, т.к. `Singer` является `Employee` (каждый экземпляр `Singer` наследует `name`)

Нисходящее приведение небезопасно, т.к. `Employee` не является `Singer` (`Employee` не нужен метод `range()`)

Рис. 13.4. Восходящее и нисходящее преобразование

Из-за выполнения неявного восходящего приведения указатель или ссылка базового класса могут ссылаться как на объект базового класса, так и на объект производного класса — что делает необходимым динамическое связывание. Такое связывание обеспечивают виртуальные методы C++.

Виртуальные функции-члены и динамическое связывание

Давайте вернемся к процессу вызова метода через ссылку или указатель. Рассмотрим следующий код:

```
BrassPlus ophelia;      // объект производного класса
Brass * bp;             // указатель на базовый класс
bp = &ophelia;          // указатель Brass на объект BrassPlus
bp->ViewAcct();         // какой вариант?
```

Как уже было сказано ранее, если функция `ViewAcct()` не объявлена как виртуальная в базовом классе, то выражение `bp->ViewAcct()` руководствуется типом ука-

зателя (`Brass *`) и вызывает `Brass::ViewAcct()`. Тип указателя известен во время компиляции, поэтому компилятор может связать `ViewAcct()` с `Brass::ViewAcct()` еще на этапе компиляции. В общем, для не виртуальных методов компилятор использует статическое связывание.

Однако если функция `ViewAcct()` объявлена в базовом классе как виртуальная, то выражение `bp->ViewAcct()` руководствуется типом объекта (`BrassPlus`) и вызывает `BrassPlus::ViewAcct()`. В этом примере видно, что тип объекта `BrassPlus`, но в общем случае (как в листинге 13.10) тип объекта может быть определен только во время выполнения. Поэтому компилятор генерирует код, который во время выполнения программы связывает `ViewAcct()` с `Brass::ViewAcct()` или `BrassPlus::ViewAcct()`, в зависимости от типа объекта. То есть для виртуальных методов компилятор использует динамическое связывание.

В большинстве случаев динамическое связывание — это хорошо, т.к. оно позволяет программе выбирать метод, предназначенный для конкретного типа. Но теперь возникают следующие вопросы.

- Зачем нужны два типа связывания?
- Если динамическое связывание такое удобное, почему оно не используется по умолчанию?
- Как работает динамическое связывание?

И сейчас мы рассмотрим ответы на эти вопросы.

Зачем существуют два типа связывания, и почему по умолчанию применяется статическое связывание

Если динамическое связывание позволяет полностью переопределять методы класса, а статическое — только частично, то зачем вообще нужно статическое связывание? На то имеются две причины: эффективность и концептуальная модель.

Сначала поговорим об эффективности. Чтобы программа могла принимать решения во время выполнения, ей надо как-то узнавать, к какому типу объекта обращается указатель или ссылка базового класса, а это требует дополнительных действий. (Ниже будет продемонстрирован один способ динамического связывания.) Если, например, вы разрабатываете класс, который заведомо не будет использоваться как базовый для наследования, то вам не нужно динамическое связывание. Оно не понадобится и тогда, когда вы используете производный класс (вроде `RatedPlayer`), который не переопределяет методы. В таких случаях имеет смысл применять статическое связывание, что слегка увеличивает эффективность. Бо́льшая эффективность статического связывания и является причиной того, что оно выбирается в C++ по умолчанию. Страуструп упоминает в связи с этим один из руководящих принципов C++: вы не должны платить (расходовать память или время) за те возможности, которые вы не используете. Поэтому к виртуальным функциям стоит прибегать только тогда, когда они нужны по сути задачи.

А теперь рассмотрим концептуальную модель. Бывает, что при разработке класса появляются функции, которые нежелательно переопределять в производных классах. Примером может служить функция `Brass::Balance()`, которая возвращает баланс счета. Объявив эту функцию невиртуальной, вы, во-первых, повысите ее эффективность, а, во-вторых, заявите, что эта функция не должна переопределяться. Значит, объявлять виртуальными следует только те методы, которые предположительно будут переопределяться.

Совет

Если метод базового класса будет переопределяться в производном классе, его необходимо объявить виртуальным. Если метод не будет переопределяться, он должен быть не виртуальным.

Конечно, во время разработки класса не всегда точно известно, в какую категорию попадает метод. Подобно многим аспектам реальной жизни, разработка классов не является линейным процессом.

Как работают виртуальные функции

Язык C++ определяет, как должны вести себя виртуальные функции, но реализация этого механизма возложена на разработчика компилятора. Чтобы использовать виртуальные функции, не нужно знать способ реализации, однако изучение принципа работы поможет ориентироваться в программах.

Обычно компиляторы управляют виртуальными функциями, добавляя в каждый объект скрытый член. Этот член хранит указатель на массив адресов функций. Такой массив обычно называется *таблицей виртуальных функций*. Таблица виртуальных функций хранит адреса виртуальных функций, объявленных для объектов данного класса.

Например, объект базового класса содержит указатель на таблицу адресов всех виртуальных функций для этого класса. Объект производного класса содержит указатель на отдельную таблицу адресов.

Если производный класс дает новое определение виртуальной функции, то в таблице виртуальных функций содержится адрес новой функции. Если производный класс не переопределяет виртуальную функцию, таблица виртуальных функций хранит адрес исходной версии функции.

Если производный класс определяет новую функцию и объявляет ее виртуальной, ее адрес добавляется в таблицу виртуальных функций (рис. 13.5). Учтите, что независимо от количества виртуальных функций, в объект добавляется только один адрес; варьируется только размер самой таблицы.

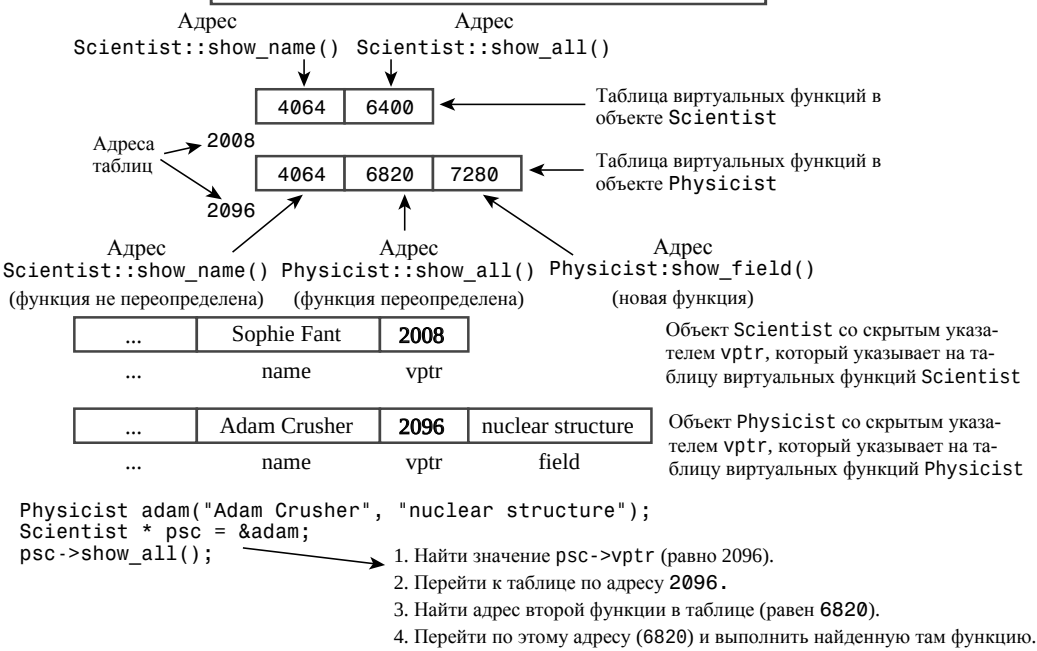
При вызове виртуальной функции программа находит адрес таблицы виртуальных функций, хранящийся в объекте, и переходит к соответствующей таблице адресов функций. Если вызывается первая виртуальная функция, определенная в объявлении класса, программа берет первый адрес в массиве и выполняет функцию с этим адресом. Если вызывается третья виртуальная функция в объявлении класса, программа выполняет функцию, адрес которой хранится в третьем элементе массива.

В общем, использование виртуальных функций приводит к следующим (небольшим) затратам памяти и снижению скорости выполнения.

- Размер каждого объекта увеличивается на значение, необходимое для хранения адреса.
- Для каждого класса компилятор создает таблицу (массив) адресов виртуальных функций.
- При каждом вызове функции выполняется дополнительный шаг поиска адреса в таблице.

Не забывайте, что не виртуальные функции слегка эффективнее виртуальных, но они не обеспечивают динамического связывания.

```
class Scientist{                          // ученый
{
    ...
    char name[40];
public:
    virtual void show_name();
    virtual void show_all();
    ...
};
class Physicist : public Scientist  // физик
{
    ...
    char field[40];
public;
    void show_all(); // переопределена
    virtual void show_field();//новая
    ...
};
```




```
Physicist adam("Adam Crusher", "nuclear structure");
Scientist * psc = &adam;
psc->show_all();
```

1. Найти значение `psc->vptr` (равно 2096).
2. Перейти к таблице по адресу `2096`.
3. Найти адрес второй функции в таблице (равен `6820`).
4. Перейти по этому адресу (`6820`) и выполнить найденную там функцию.

Рис. 13.5. Механизм работы виртуальных функций

Что следует знать о виртуальных методах

Мы уже обсудили основные моменты, связанные с виртуальными методами.

- **Если в базовом классе объявление метода класса начинается с ключевого слова `virtual`, то функция становится виртуальной для базового класса и для всех классов, производных от данного, включая класса, порожденные от порожденных классов, и т.д.**
- **Если виртуальный метод вызывается через ссылку или указатель на объект, то программа использует метод, определенный для типа объекта, а не для типа указателя или ссылки. Это называется *динамическим* (или *поздним*) *связыванием*. Такое поведение очень важно, т.к. указатель или ссылка на базовый класс всегда может обратиться к объекту производного типа.**
- **При определении класса, который будет использоваться в качестве базового для наследования, следует объявить виртуальными те методы класса, которые могут быть переопределены в производных классах.**

Существует еще несколько моментов, которые необходимо знать о виртуальных методах. Некоторые из них уже были кратко упомянуты. Рассмотрим их подробнее.

Конструкторы

Конструкторы не могут быть виртуальными. При создании объекта производного класса вызывается конструктор производного, а не базового класса. Правда, затем конструктор производного класса использует конструктор базового класса, однако, эта последовательность отличается от механизма наследования. Таким образом, производный класс не наследует конструкторы базового класса, и нет смысла делать их виртуальными.

Деструкторы

Деструкторы должны быть виртуальными, за исключением тех классов, которые не используются в качестве базовых. Например, предположим, что `Employee` — это базовый класс, а `Singer` — производный класс, добавляющий член `char *`, который указывает на память, выделенную операцией `new`. Когда объект `Singer` завершает свою работу, необходимо вызвать деструктор `~Singer()`, чтобы освободить эту память.

Теперь рассмотрим следующий код:

```
Employee * pe = new Singer; //допустимо, т.к. Employee — базовый класс для Singer
...
delete pe;                  // ~Employee() или ~Singer()?
```

Если применяется стандартное статическое связывание, то оператор `delete` вызывает деструктор `~Employee()`. При этом освобождается память, на которую указывают компоненты `Employee` объекта `Singer`, но не память, на которую указывают новые члены класса. Но если деструкторы виртуальные, то тот же самый код вызывает деструктор `~Singer()` для освобождения памяти, на которую указывает компонент `Singer`, а затем вызывает деструктор `~Employee()` для освобождения памяти, на которую указывает компонент `Employee`.

Учтите, что даже если для базового класса не требуется явный деструктор, не стоит полагаться на деструктор по умолчанию. Следует указать виртуальный деструктор, даже если он ничего не будет делать:

```
virtual ~BaseClass() { }
```

Кстати, наличие виртуального деструктора не будет ошибкой, даже если вы не планируете сделать класс базовым, хотя при этом слегка пострадает эффективность.

> **Совет**
>
> Базовый класс рекомендуется снабжать виртуальным деструктором, даже если необходимость в нем отсутствует.

Дружественные функции

Друзья не могут быть виртуальными функциями: ведь они не являются членами класса, а виртуальными функциями могут быть только члены. Если это приводит к проблемам при разработке, то их можно устранить, введя виртуальные функции-члены внутри дружественных функций.

Отсутствие переопределения

Если в производном классе нет переопределения какой-то функции (виртуальной или нет), то класс будет использовать версию функции из базового класса. Если производный класс является частью длинной цепочки порождений, то будет применяться

самая последняя версия функции. Исключение составляет случай, когда базовая версия скрыта, как описано ниже.

Переопределение скрывает методы

Предположим, что написан примерно такой код:

```
class Dwelling                    // жилище
{
public:
   virtual void showperks(int a) const;
   ...
};
class Hovel : public Dwelling     // хибара
{
public:
   virtual void showperks() const;
   ...
};
```

В этом случае вы можете получить предупреждение компилятора наподобие

```
Warning: Hovel::showperks(void) hides Dwelling::showperks(int)
Внимание: Hovel::showperks(void) скрывает Dwelling::showperks(int)
```

Но, возможно, предупреждение не будет выдано. В любом случае, из приведенных определений следует:

```
Hovel trump;
trump.showperks();                // верно
trump.showperks(5);               // неверно
```

Новое определение создает функцию `showperks()`, которая не принимает аргументы. Вместо того чтобы привести к появлению двух перегруженных версий функции, это переопределение *скрывает* версию базового класса, которая принимает аргумент `int`. В общем, переопределение унаследованных методов не является разновидностью перегрузки. При переопределении функции в производном классе происходит не просто перегрузка объявления базового класса с той же самой сигнатурой функции. Вместо этого скрываются *все* методы базового класса с тем же именем и любыми сигнатурами аргументов.

Отсюда пара важных правил. Во-первых, при переопределении унаследованного метода необходимо удостовериться в точном совпадении с исходным прототипом. Одно сравнительно новое исключение из этого правила состоит в том, что если возвращаемый тип является указателем или ссылкой на базовый класс, то его можно заменить указателем или ссылкой на производный класс. Это свойство называется *ковариантностью возвращаемого типа*, поскольку возвращаемый тип можно изменять параллельно с типом класса:

```
class Dwelling                    // жилище
{
public:
// Базовый метод
   virtual Dwelling & build(int n);
   ...
};
class Hovel : public Dwelling     // хибара
{
public:
```

```
    // Производный метод с ковариантным возвращаемым типом
    virtual Hovel & build(int n);    // та же сигнатура функции
    ...
};
```

Учтите, что данное исключение относится только к возвращаемым значениям, но не к аргументам.

Во-вторых, если объявление базового класса перегружается, в производном классе необходимо переопределить все версии базового класса:

```
class Dwelling                      // жилище
{
public:
    // Три перегруженных функции showperks()
    virtual void showperks(int a) const;
     virtual void showperks(double x) const;
    virtual void showperks() const;
    ...
};
class Hovel : public Dwelling       // хибара
{
public:
    // Три переопределенных функции showperks()
    virtual void showperks(int a) const;
    virtual void showperks(double x) const;
    virtual void showperks() const;
    ...
};
```

Если переопределить только одну версию, то две остальных становятся скрытыми и не могут использоваться объектами производного класса. Если никакие изменения не нужны, то переопределение может просто вызывать версию базового класса:

```
void Hovel::showperks() const {Dwelling::showperks();}
```

Управление доступом: `protected`

До настоящего времени для управления доступом к членам наших классов использовались ключевые слова `public` и `private`. Имеется еще одна категория доступа, обозначаемая ключевым словом `protected` (защищенный).

Ключевое слово `protected` подобно `private` в том смысле, что доступ к членам класса из раздела `protected` можно получить извне только с помощью открытых членов класса. Различие между `private` и `protected` проявляется только внутри классов, порожденных от базового класса. Члены производного класса имеют прямой доступ к защищенным членам базового класса, но не имеют прямого доступа к закрытым членам базового класса. То есть члены из защищенной категории ведут себя как закрытые члены для внешнего мира и как открытые члены для производных классов.

Например, предположим, что в классе `Brass` член `balance` объявлен как `protected`:

```
class Brass
{
protected:
    double balance;
    ...
};
```

В этом случае класс BrassPlus имеет прямой доступ к члену balance без применения методов Brass. Например, ядро функции BrassPlus::Withdraw() можно записать так:

```
void BrassPlus::Withdraw(double amt)
{
   if (amt < 0)
      cout << "Withdrawal amount must be positive; "
       << "withdrawal canceled.\n"; // снимаемая сумма должна быть положительной
   else if (amt <= balance)        // прямой доступ к balance
      balance -= amt;
   else if ( amt <= balance + maxLoan - owesBank)
   {
      double advance = amt - balance;
      owesBank += advance * (1.0 + rate);
      cout << "Bank advance: $" << advance << endl;          // аванс банка
      cout << "Finance charge: $" << advance * rate << endl; // долг банку
      Deposit(advance);
      balance -= amt;
   }
   else
      cout << "Credit limit exceeded. Transaction cancelled.\n";
}
```

Защищенные данные-члены могут упростить код, но в нем присутствует проектный изъян. Например, если бы член balance в классе BrassPlus был защищенным, то код можно было бы записать следующим образом:

```
void BrassPlus::Reset(double amt)
{
   balance = amt;
}
```

Класс Brass разработан таким образом, что интерфейс функций Deposit() и Withdraw() предусматривает только один способ для изменения balance. Однако метод Reset(), по сути, делает balance открытой переменной для объектов BrassPlus, обходя, например, защитные меры в функции Withdraw().

Внимание!

При работе с данными-членами класса старайтесь использовать защищенный доступ, а не закрытый, а для доступа из производных классов к данным базового класса применяйте методы базового класса.

Однако защищенный доступ может оказаться достаточно полезным для функций-членов, предоставляя производным классам доступ к внутренним функциям, которые не являются открытыми.

Абстрактные базовые классы

Мы уже знакомы с простым наследованием и более сложным полиморфным наследованием. Следующий шаг по увеличению сложности — абстрактный базовый класс (АБК). Рассмотрим некоторые ситуации, которые лежат в основе концепции АБК.

Иногда использование отношения *является* не настолько просто, как может показаться. Предположим, например, что вы разрабатываете графическую программу, которая должна выводить среди прочих объектов окружности и эллипсы. Окружность представляет собой частный случай эллипса: это эллипс, у которого бо́льшая полуось

равна меньшей. И поскольку все окружности являются эллипсами, заманчиво породить класс Circle от класса Ellipse. Однако когда дело дойдет до реализации, могут возникнуть проблемы.

Чтобы убедиться в этом, сначала нужно решить, что должно входить в класс Ellipse. Данными-членами могут быть координаты центра, большая полуось (половина большего диаметра), малая полуось (половина меньшего диаметра) и наклон — угол между горизонтальной осью координат и большой полуосью. Также в класс могут входить методы для перемещения эллипса, вычисления площади, вращения и для растягивания большой и малой полуосей:

```
class Ellipse
{
private:
   double x;      // координата x центра эллипса
   double y;      // координата y центра эллипса
   double a;      // большая полуось
   double b;      // малая полуось
   double angle;  // угол наклона в градусах
   ...
public:
   ...
   void Move(int nx, ny) { x = nx; y = ny; }
   virtual double Area() const { return 3.14159 * a * b; }
   virtual void Rotate(double nang) { angle += nang; }
   virtual void Scale(double sa, double sb) { a *= sa; b *= sb; }
   ...
};
```

Теперь предположим, что класс Circle порождается от класса Ellipse:

```
class Circle : public Ellipse
{
   ...
};
```

Хотя окружность и является эллипсом, такое порождение несколько неуклюже. Например, размер и форма окружности задаются только одним значением (ее радиусом); для нее не нужны величины большой полуоси (a) и малой полуоси (b). Конструкторы Circle могут присвоить одно и то же значение членам a и b, но тогда будет избыточное представление одной и той же информации. Параметр angle и метод Rotate() не имеют смысла для окружности, а метод Scale() в существующем виде может превратить окружность в овал, по-разному растянув две оси. Можно попробовать устранить эти проблемы с помощью различных ухищрений — например, поместить переопределенный метод Rotate() в закрытый раздел класса Circle, чтобы он стал недоступным для окружности. Однако в целом легче определить класс Circle без применения наследования:

```
class Circle   // без наследования
{
private:
   double x;   // координата x центра окружности
   double y;   // координата y центра окружности
   double r;   // радиус
   ...
public:
   ...
```

```
    void Move(int nx, ny) { x = nx; y = ny; }
    double Area() const { return 3.14159 * r * r; }
    void Scale(double sr) { r *= sr; }
    ...
};
```

Теперь класс содержит только необходимые переменные. Но это решение тоже не удовлетворительно. Классы `Circle` и `Ellipse` имеют много общего, однако при их отдельном определении этот факт игнорируется.

Существует другое решение. Из классов `Ellipse` и `Circle` можно извлечь их общие свойства и поместить их в АБК. Затем можно породить от этого АБК и `Circle`, и `Ellipse`. После этого можно, например, использовать массив указателей базового класса для работы со смесью объектов `Ellipse` и `Circle` — т.е. воспользоваться полиморфизмом. В данном случае для двух классов общими являются координаты центра фигуры, метод `Move()`, который работает одинаково для двух классов, а также метод `Area()`, работающий по-разному. Вообще говоря, метод `Area()` и невозможно реализовать для АБК, т.к. в нем нет необходимых данных-членов. В C++ имеется способ для представления нереализованной функции — *чистая виртуальная функция*. Чистая виртуальная функция в конце своего объявления содержит конструкцию `= 0`, как, например, в следующем методе `Area()`:

```
class BaseEllipse    // абстрактный базовый класс
{
private:
    double x;        // координата x центра
    double y;        // координата y центра
    ...
public:
    BaseEllipse(double x0 = 0, double y0 = 0) : x(x0), y(y0) {}
    virtual ~BaseEllipse() {}
    void Move(int nx, ny) { x = nx; y = ny; }
    virtual double Area() const = 0; // чистая виртуальная функция
    ...
}
```

Если объявление класса содержит чистую виртуальную функцию, то объект такого класса создать невозможно. Смысл классов с чистыми виртуальными функциями в том, что они предназначены только для использования в качестве базовых классов. Чтобы класс был настоящим АБК, он должен содержать, по крайней мере, одну чистую виртуальную функцию. Обычная виртуальная функция превращается в чистую с помощью конструкции `= 0` в прототипе. В случае метода `Area()` функция не имеет определения, но в C++ даже для чистой виртуальной функции допускается иметь определение. Например, возможно, что все базовые методы похожи на `Move()` тем, что они могут быть определены для базового класса, но класс все-таки нужно сделать абстрактным. Тогда можно сделать абстрактным прототип:

```
void Move(int nx, ny) = 0;
```

Базовый класс при этом становится абстрактным. Но после этого все равно можно записать определение в файле реализации:

```
void BaseEllipse::Move(int nx, ny) { x = nx; y = ny; }
```

В общем, конструкция `= 0` в прототипе указывает, что класс является абстрактным базовым классом, и функцию в нем определять не обязательно.

Теперь от класса BaseEllipse можно породить классы Ellipse и Circle, добавляя члены, необходимые для завершения каждого класса. Один важный момент состоит в том, что класс Circle всегда представляет окружности, а класс Ellipse представляет эллипсы, которые могут быть и окружностями. Однако окружность класса Ellipse можно трансформировать в эллипс, а окружность класса Circle должна оставаться окружностью.

Программа, использующая эти классы, сможет создавать объекты Ellipse и Circle, но не BaseEllipse. Поскольку у объектов Circle и Ellipse один и тот же базовый класс, с коллекцией таких объектов можно работать с помощью массива указателей на BaseEllipse. Классы, подобные Circle и Ellipse, иногда называются *конкретными* классами, чтобы подчеркнуть возможность создавать объекты данных типов.

Итак, АБК описывает интерфейс, содержащий, по меньшей мере, одну чистую виртуальную функцию. Классы, порождаемые от АБК, содержат обычные виртуальные функции для реализации интерфейса со свойствами конкретного производного класса.

Применение концепции абстрактных базовых классов

Вы, возможно, хотели бы увидеть полный пример АБК, поэтому давайте применим этот принцип к представлению счетов Brass и BrassPlus, начав с абстрактного базового класса AcctABC. Этот класс должен содержать все методы и данные-члены, общие для классов Brass и BrassPlus. Методы, работа которых различается для классов BrassPlus и Brass, необходимо объявлять как виртуальные функции. Чтобы сделать класс AcctABC абстрактным, по крайней мере, одна виртуальная функция должна быть чистой виртуальной.

В листинге 13.11 приведен заголовочный файл, в котором объявлен класс AcctABC (абстрактный базовый класс), а также классы Brass и BrassPlus (конкретные классы). Для облегчения доступа производного класса к данным базового класса в AcctABC имеется несколько защищенных методов. Вспомните, что защищенные методы – это методы, которые может вызывать производный класс, однако они не входят в общедоступный интерфейс для объектов производного класса. Кроме того, класс AcctABC предоставляет защищенную функцию-член для управления форматированием, которое ранее выполнялось в сторонних функциях. В классе AcctABC имеются две чистые виртуальные функции, поэтому он, несомненно, является абстрактным.

Листинг 13.11. acctabc.h

```
// acctabc.h -- классы банковских счетов
#ifndef ACCTABC_H_
#define ACCTABC_H_
#include <iostream>
#include <string>

// Абстрактный базовый класс
class AcctABC
{
private:
    std::string fullName;
    long acctNum;
    double balance;
protected:
    struct Formatting
    {
        std::ios_base::fmtflags flag;
        std::streamsize pr;
    };
```

```
    const std::string & FullName() const {return fullName;}
    long AcctNum() const {return acctNum;}
    Formatting SetFormat() const;
    void Restore(Formatting & f) const;
public:
    AcctABC(const std::string & s = "Nullbody", long an = -1,
            double bal = 0.0);
    void Deposit(double amt) ;
    virtual void Withdraw(double amt) = 0;     // чистая виртуальная функция
    double Balance() const {return balance;};
    virtual void ViewAcct() const = 0;         // чистая виртуальная функция
    virtual ~AcctABC() {}
};

// Класс счета Brass Account
class Brass :public AcctABC
{
public:
    Brass(const std::string & s = "Nullbody", long an = -1,
          double bal = 0.0) : AcctABC(s, an, bal) { }
    virtual void Withdraw(double amt);
    virtual void ViewAcct() const;
    virtual ~Brass() {}
};

// Класс счета Brass Plus
class BrassPlus : public AcctABC
{
private:
    double maxLoan;
    double rate;
    double owesBank;
public:
    BrassPlus(const std::string & s = "Nullbody", long an = -1,
              double bal = 0.0, double ml = 500,
              double r = 0.10);
    BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);
    virtual void ViewAcct()const;
    virtual void Withdraw(double amt);
    void ResetMax(double m) { maxLoan = m; }
    void ResetRate(double r) { rate = r; };
    void ResetOwes() { owesBank = 0; }
};
#endif
```

Следующий шаг — реализация методов, у которых нет встроенных определений. Это сделано в листинге 13.12.

Листинг 13.12. acctabc.cpp

```
// acctabc.cpp -- методы класса банковских счетов
#include <iostream>
#include "acctabc.h"
using std::cout;
using std::ios_base;
using std::endl;
using std::string;
```

```
// Абстрактный базовый класс
AcctABC::AcctABC(const string & s, long an, double bal)
{
   fullName = s;
   acctNum = an;
   balance = bal;
}

void AcctABC::Deposit(double amt)
{
   if (amt < 0)
      cout << "Negative deposit not allowed; "
           << "deposit is cancelled.\n";  // отрицательный вклад не допускается
   else
      balance += amt;
}

void AcctABC::Withdraw(double amt)
{
   balance -= amt;
}

// Защищенные методы для форматирования
AcctABC::Formatting AcctABC::SetFormat() const
{
   // Установка формата ###.##
   Formatting f;
   f.flag =
      cout.setf(ios_base::fixed, ios_base::floatfield);
   f.pr = cout.precision(2);
   return f;
}

void AcctABC::Restore(Formatting & f) const
{
   cout.setf(f.flag, ios_base::floatfield);
   cout.precision(f.pr);
}

// Методы Brass
void Brass::Withdraw(double amt)
{
   if (amt < 0)
      cout << "Withdrawal amount must be positive; "
           << "withdrawal canceled.\n"; // снимаемая сумма должна быть положительной
   else if (amt <= Balance())
      AcctABC::Withdraw(amt);
   else
      cout << "Withdrawal amount of $" << amt
           << " exceeds your balance.\n"
           << "Withdrawal canceled.\n"; // снимаемая сумма превышает текущий баланс
}

void Brass::ViewAcct() const
{
   Formatting f = SetFormat();
   cout << "Brass Client: " << FullName() << endl;      // клиент Brass
   cout << "Account Number: " << AcctNum() << endl;     // номер счета
   cout << "Balance: $" << Balance() << endl;           // баланс
   Restore(f);
}
```

```
// Методы BrassPlus
BrassPlus::BrassPlus(const string & s, long an, double bal,
                 double ml, double r) : AcctABC(s, an, bal)
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}
BrassPlus::BrassPlus(const Brass & ba, double ml, double r)
       : AcctABC(ba) // используется неявный конструктор копирования
{
   maxLoan = ml;
   owesBank = 0.0;
   rate = r;
}
void BrassPlus::ViewAcct() const
{
   Formatting f = SetFormat();
   cout << "BrassPlus Client: " << FullName() << endl;  // клиент BrassPlus
   cout << "Account Number: " << AcctNum() << endl;     // номер счета
   cout << "Balance: $" << Balance() << endl;           // баланс
   cout << "Maximum loan: $" << maxLoan << endl;        // максимальный заем
   cout << "Owed to bank: $" << owesBank << endl;       // долг банку
   cout.precision(3);
   cout << "Loan Rate: " << 100 * rate << "%\n";        // процент на заем
   Restore(f);
}
void BrassPlus::Withdraw(double amt)
{
   Formatting f = SetFormat();
   double bal = Balance();
   if (amt <= bal)
      AcctABC::Withdraw(amt);
   else if ( amt <= bal + maxLoan - owesBank)
   {
      double advance = amt - bal;
      owesBank += advance * (1.0 + rate);
      cout << "Bank advance: $" << advance << endl;              // аванс банка
      cout << "Finance charge: $" << advance * rate << endl;     // долг банку
      Deposit(advance);
      AcctABC::Withdraw(amt);
   }
   else
    cout << "Credit limit exceeded. Transaction cancelled.\n"; //предел кредита превышен
   Restore(f);
}
```

Защищенные методы `FullName()` и `AcctNum()` предоставляют доступ только для чтения к членам данных `fullName` и `acctNum`, а также позволяют индивидуально настроить функцию `ViewAcct()` для каждого производного класса.

В этой версии содержится пара усовершенствований в форматировании. В предыдущей версии использовались два вызова функции для указания форматирования и один вызов для восстановления:

```
format initialState = setFormat();
precis prec = cout.precision(2);
...
restore(initialState, prec);  // восстановление исходного формата
```

В новой версии определена структура для хранения двух значений форматирования, которая применяется для указания и восстановления форматов лишь за два вызова:

```
struct Formatting
{
   std::ios_base::fmtfglas flag;
   std::streamsize pr;
};
...
Formatting f = SetFormat();
...
Restore(f);
```

Это выглядит аккуратнее.

Проблема со старой версией была в том, что в ней функции `setFormat()` и `restore()` были автономными, и их имена могли конфликтовать с именами функций, определенных клиентом. Устранить такую проблему можно несколькими способами. Один из них — объявление обеих функций с квалификатором `static`, что делает их закрытыми в файле `brass.cpp` или его предшественнике — файле `acctabc.cpp`. Второй способ — помещение обеих функций и определения `struct Formatting` в пространство имен. Но одной из тем в рассматриваемом примере является защищенный доступ, поэтому здесь определения структур и функции помещены в защищенную часть определения класса. Это делает их доступными в базовом классе и производных классах, но скрывает от внешнего мира.

Новую реализацию счетов `Brass` и `BrassPlus` можно использовать таким же образом, как и старую, поскольку у методов класса те же имена и интерфейсы, что и ранее. Например, чтобы преобразовать код в листинге 13.10 для применения новой реализации (преобразовать файл `usebrass2.cpp` в `usebrass3.cpp`), необходимо выполнить перечисленные ниже шаги.

- Связать `usebrass2.cpp` с `acctabc.cpp` вместо `brass.cpp`.
- Включить `acctabc.h` вместо `brass.h`.
- Заменить

  ```
  Brass * p_clients[CLIENTS];
  ```

  на

  ```
  AcctABC * p_clients[CLIENTS];
  ```

Полученный файл приведен в листинге 13.13 с новым именем `usebrass3.cpp`.

Листинг 13.13. `usebrass3.cpp`

```
// usebrass3.cpp — полиморфный пример с использованием абстрактного базового класса
// Компилировать вместе с acctacb.cpp
#include <iostream>
#include <string>
#include "acctabc.h"
const int CLIENTS = 4;
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   AcctABC * p_clients[CLIENTS];
   std::string temp;
```

```
    long tempnum;
    double tempbal;
    char kind;
    for (int i = 0; i < CLIENTS; i++)
    {
        cout << "Enter client's name: ";              // ввод имени клиента
        getline(cin,temp);
        cout << "Enter client's account number: ";    // ввод номера счета клиента
        cin >> tempnum;
        cout << "Enter opening balance: $";           // ввод начального баланса
        cin >> tempbal;
        cout << "Enter 1 for Brass Account or "       // 1 -- Brass Account;
             << "2 for BrassPlus Account: ";          // 2 -- BrassPlus Account
        while (cin >> kind && (kind != '1' && kind != '2'))
            cout <<"Enter either 1 or 2: ";
        if (kind == '1')
            p_clients[i] = new Brass(temp, tempnum, tempbal);
        else
        {
            double tmax, trate;
            cout << "Enter the overdraft limit: $";  // ввод предела овердрафта
            cin >> tmax;
            cout << "Enter the interest rate "
                 << "as a decimal fraction: ";       // ввод процентной ставки
            cin >> trate;
            p_clients[i] = new BrassPlus(temp, tempnum, tempbal, tmax, trate);
        }
        while (cin.get() != '\n')
            continue;
    }
    cout << endl;
    for (int i = 0; i < CLIENTS; i++)
    {
        p_clients[i]->ViewAcct();
        cout << endl;
    }
    for (int i = 0; i < CLIENTS; i++)
    {
        delete p_clients[i];                          // освобождение памяти
    }
    cout << "Done.\n";
    return 0;
}
```

Эта программа ведет себя точно так же, как и версия без АБК, поэтому при одинаковых входных данных будет получен тот же вывод, что и для листинга 13.10.

Философия АБК

Методология АБК представляет собой гораздо более систематический и упорядоченный подход к наследованию по сравнению с конкретным ситуационным принципом, который использован в примере с `RatedPlayer`. Прежде чем приступить к разработке АБК, сначала нужно выяснить, какие классы необходимы в данной задаче, и как они зависят друг от друга. Одна из концепций заключается в том, что при проектировании иерархии наследования классов конкретными классами должны быть только те, которые никогда не будут выступать в качестве базовых классов. Такой подход позволяет получить более ясные конструкции с меньшими затратами.

Абстрактные базовые классы можно рассматривать как способ введение интерфейса. АБК требует, чтобы его чистые виртуальные функции перегружались во всех конкретных производных классах – т.е. производный класс должен подчиняться правилам интерфейса, установленным в АБК. Эта модель общепринята в парадигмах программирования на основе компонентов, где применение АБК позволяет разработчику компонентов создавать "интерфейсный контракт". Так гарантируется, что все компоненты, порожденные от АБК, поддерживают, по меньшей мере, общие возможности, установленные АБК.

Наследование и динамическое выделение памяти

Как наследование соотносится с динамическим распределением памяти (операциями new и delete)? Например, если базовый класс применяет динамическое выделение памяти и перегружает операцию присваивания и конструктор копирования, то каким образом это отражается на реализации производного класса? Ответ зависит от природы производного класса. Если сам производный класс не использует динамическое выделение памяти, то никакие особые меры не нужны. Но если использует, придется освоить несколько новых приемов. Рассмотрим эти два случая.

Случай 1: производный класс не использует операцию new

Допустим, имеется следующий базовый класс, в котором используется динамическое выделение памяти:

```
// Базовый класс, использующий динамическое выделение памяти
class baseDMA
{
private:
    char * label;
    int rating;
public:
    baseDMA(const char * l = "null", int r = 0);
    baseDMA(const baseDMA & rs);
    virtual ~baseDMA();
    baseDMA & operator=(const baseDMA & rs);
    ...
};
```

Это объявление содержит специальные методы, необходимые, когда в конструкторах применяется операция new – деструктор, конструктор копирования и перегруженную операцию присваивания.

Теперь предположим, что от класса baseDMA нужно породить класс lackDMA, в котором не используется ни операция new, ни другие нестандартные возможности, которые требуют особого обращения:

```
// Производный класс, не использующий динамическое выделение памяти
class lacksDMA :public baseDMA
{
private:
    char color[40];
public:
    ...
};
```

Нужно ли определять явный деструктор, конструктор копирования и операцию присваивания для класса lackDMA? Ответ отрицательный.

Сначала посмотрим, необходим ли деструктор. Если он не определен, компилятор генерирует деструктор по умолчанию, который ничего не делает. Вообще говоря, деструктор по умолчанию производного класса всегда что-то делает: после выполнения собственного кода он вызывает деструктор базового класса. Поскольку по предположению члены lackDMA не требуют особых действий, то деструктор по умолчанию вполне годится.

Теперь рассмотрим конструктор копирования. Как было показано в главе 12, конструктор копирования по умолчанию выполняет почленное копирование, что неприемлемо при динамическом выделении памяти. Однако почленное копирование годится для нового члена lacksDMA, поскольку не искажает унаследованный объект baseDMA. Нужно помнить, что почленное копирование использует форму копирования, которая определена для конкретного типа данных. Поэтому для копирования long в long осуществляется обычное присваивание. Однако копирование члена класса или унаследованного компонента класса выполняется конструктором копирования для данного класса. Поэтому конструктор копирования по умолчанию для класса lacksDMA использует явный конструктор копирования baseDMA, чтобы скопировать часть baseDMA из объекта lacksDMA. Значит, конструктор копирования по умолчанию годится для нового члена lacksDMA, а также для унаследованного объекта baseDMA.

Практически все это верно и для присваивания. Операция присваивания по умолчанию для производного класса автоматически выполняет операцию присваивания базового класса для компонента базового класса. Значит, и здесь все нормально.

Эти свойства унаследованных объектов верны и для членов класса, которые сами являются объектами. Например, в главе 10 в реализации класса Stock для представления названия компании использовался объект string. Стандартный класс string, как и наш пример String, использует динамическое выделение памяти. Теперь вы уже знаете, почему это не вызывает проблем. Конструктор копирования по умолчанию класса Stock будет применять конструктор копирования string для копирования члена company объекта. Операция присваивания по умолчанию класса Stock будет использовать операцию присваивания string для присваивания значения члену company объекта. Деструктор Stock (по умолчанию или какой-то другой) будет автоматически вызывать деструктор string.

Случай 2: производный класс использует операцию new

Предположим, что в производном классе применяется операция new:

```
// Производный класс, использующий динамическое выделение памяти
class hasDMA :public baseDMA
{
private:
    char * style; // использование new в конструкторах
public:
    ...
};
```

В этом случае, конечно, для производного класса необходимо определить явный деструктор, конструктор копирования и операцию присваивания. Рассмотрим эти методы по очереди.

Деструктор производного класса автоматически вызывает деструктор базового класса, поэтому он сам отвечает только за зачистку действий конструкторов произ-

водного класса. Значит, деструктор hasDMA должен освободить память, управляемую указателем style, и передать управление деструктору baseDMA, который освободит память, управляемую указателем label:

```
baseDMA::~baseDMA()  // очистка в baseDMA
{
   delete [] label;
}
hasDMA::~hasDMA()    // очистка в hasDMA
{
   delete [] style;
}
```

Теперь рассмотрим конструкторы копирования. Конструктор копирования baseDMA следует обычной модели для символьных массивов. Это значит, что с помощью функции strlen() определяется размер памяти, необходимой для хранения строки в стиле C, выделяется достаточный объем памяти (количество символов плюс один байт для нулевого символа) и используется функция strcpy() для копирования исходной строки:

```
baseDMA::baseDMA(const baseDMA & rs)
{
   label = new char[std::strlen(rs.label) + 1];
   std::strcpy(label, rs.label);
   rating = rs.rating;
}
```

Конструктор копирования hasDMA имеет доступ только к данным hasDMA, поэтому он должен вызвать конструктор копирования baseDMA для обработки части данных baseDMA:

```
hasDMA::hasDMA(const hasDMA & hs) : baseDMA(hs)
{
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
}
```

Здесь важно то, что список инициализаторов членов передает ссылку hasDMA конструктору baseDMA. Не существует ни одного конструктора baseDMA с параметром типа ссылки на hasDMA, но они и не нужны: ведь конструктору копирования baseDMA передается ссылка на baseDMA, а ссылка на базовый класс может ссылаться и на производный класс. Поэтому конструктор копирования baseDMA использует порцию baseDMA аргумента hasDMA для создания порции baseDMA нового объекта.

Теперь рассмотрим операции присваивания. Операция присваивания baseDMA выглядит вполне обычно:

```
baseDMA & baseDMA::operator=(const baseDMA & rs)
{
   if (this == &rs)
      return *this;
   delete [] label;
   label = new char[std::strlen(rs.label) + 1];
   std::strcpy(label, rs.label);
   rating = rs.rating;
   return *this;
}
```

Поскольку класс hasDMA также использует динамическое выделение памяти, в нем нужна явная операция присваивания. Будучи методом hasDMA, он может непосредственно обращаться только к данным hasDMA. Но явная операция присваивания для производного класса должна позаботиться и о присваивании для унаследованного объекта baseDMA базового класса. Это можно сделать с помощью явного вызова операции присваивания, определенной в базовом классе:

```
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
   if (this == &hs)
      return *this;
   baseDMA::operator=(hs);      // копирование базовой части
   delete [] style;             // подготовка к операции new для style
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
   return *this;
}
```

Показанный ниже оператор может выглядеть несколько непривычно:

```
baseDMA::operator=(hs);         // копирование базовой части
```

Однако применение функциональной, а не операционной, нотации позволяет использовать операцию разрешения контекста. В сущности, этот оператор означает следующее:

```
*this = hs;                     // использовать baseDMA::operator=()
```

Разумеется, компилятор игнорирует комментарии, поэтому из последнего кода компилятор сформирует оператор hasDMA::operator=() — т.е. рекурсивный вызов. А вот использование функциональной нотации приводит к вызову нужной операции присваивания.

Подведем итоги. Если и базовый, и производный классы используют динамическое выделение памяти, то деструктор, конструктор копирования и операция присваивания производного класса должны применять свои аналоги из базового класса для обработки компонента базового класса. Это обычное требование удовлетворяется тремя различными способами. Для деструктора оно выполняется автоматически. Для конструктора — с помощью вызова конструктора копирования базового класса в списке инициализаторов членов либо автоматического вызова конструктора по умолчанию. Для операции присваивания — посредством операции разрешения контекста в явном вызове операции присваивания базового класса.

Пример наследования с динамическим выделением памяти и дружественными функциями

Для иллюстрации концепций наследования и динамического выделения памяти давайте объединим рассмотренные выше классы baseDMA, lacksDMA и hasDMA в один пример. В листинге 13.14 приведен заголовочный файл для этих классов. Кроме всего уже рассмотренного, в нем добавлена дружественная функция — для демонстрации того, как производные классы могут получать доступ к друзьям базового класса.

Листинг 13.14. dma.h

```
// dma.h -- наследование и динамическое выделение памяти
#ifndef DMA_H_
#define DMA_H_
#include <iostream>
```

```
// Базовый класс, использующий динамическое выделение памяти
class baseDMA
{
private:
   char * label;
   int rating;
public:
   baseDMA(const char * l = "null", int r = 0);
   baseDMA(const baseDMA & rs);
   virtual ~baseDMA();
   baseDMA & operator=(const baseDMA & rs);
   friend std::ostream & operator<<(std::ostream & os,
   const baseDMA & rs);
};
// Производный класс без динамического выделения памяти
// Деструктор не нужен
// Используется неявный конструктор копирования
// Используется неявная операция присваивания
class lacksDMA :public baseDMA
{
private:
   enum { COL_LEN = 40};
   char color[COL_LEN];
public:
   lacksDMA(const char * c = "blank", const char * l = "null",
          int r = 0);
   lacksDMA(const char * c, const baseDMA & rs);
   friend std::ostream & operator<<(std::ostream & os,
                                    const lacksDMA & rs);
};
// Производный класс с динамическим выделением памяти
class hasDMA :public baseDMA
{
private:
   char * style;
public:
   hasDMA(const char * s = "none", const char * l = "null",
         int r = 0);
   hasDMA(const char * s, const baseDMA & rs);
   hasDMA(const hasDMA & hs);
   ~hasDMA();
   hasDMA & operator=(const hasDMA & rs);
   friend std::ostream & operator<<(std::ostream & os,
                                    const hasDMA & rs);
};
#endif
```

В листинге 13.15 приведены определения методов для классов `baseDMA`, `lacksDMA` и `hasDMA`.

Листинг 13.15. `dma.cpp`

```
// dma.cpp -- методы классов с динамическим выделением памяти
#include "dma.h"
#include <cstring>
```

```
// Методы baseDMA
baseDMA::baseDMA(const char * l, int r)
{
   label = new char[std::strlen(l) + 1];
   std::strcpy(label, l);
   rating = r;
}
baseDMA::baseDMA(const baseDMA & rs)
{
   label = new char[std::strlen(rs.label) + 1];
   std::strcpy(label, rs.label);
   rating = rs.rating;
}
baseDMA::~baseDMA()
{
   delete [] label;
}
baseDMA & baseDMA::operator=(const baseDMA & rs)
{
   if (this == &rs)
      return *this;
   delete [] label;
   label = new char[std::strlen(rs.label) + 1];
   std::strcpy(label, rs.label);
   rating = rs.rating;
   return *this;
}
std::ostream & operator<<(std::ostream & os, const baseDMA & rs)
{
   os << "Label: " << rs.label << std::endl;       // название
   os << "Rating: " << rs.rating << std::endl;     // рейтинг
   return os;
}
// Методы lacksDMA
lacksDMA::lacksDMA(const char * c, const char * l, int r)
      : baseDMA(l, r)
{
   std::strncpy(color, c, 39);
   color[39] = '\0';
}
lacksDMA::lacksDMA(const char * c, const baseDMA & rs)
      : baseDMA(rs)
{
   std::strncpy(color, c, COL_LEN - 1);
   color[COL_LEN - 1] = '\0';
}
std::ostream & operator<<(std::ostream & os, const lacksDMA & ls)
{
   os << (const baseDMA &) ls;
   os << "Color: " << ls.color << std::endl;       // цвет
   return os;
}
// Методы hasDMA
hasDMA::hasDMA(const char * s, const char * l, int r)
    : baseDMA(l, r)
{
   style = new char[std::strlen(s) + 1];
   std::strcpy(style, s);
}
```

```
hasDMA::hasDMA(const char * s, const baseDMA & rs)
    : baseDMA(rs)
{
   style = new char[std::strlen(s) + 1];
   std::strcpy(style, s);
}
hasDMA::hasDMA(const hasDMA & hs)
    : baseDMA(hs)              // вызывает конструктор копирования базового класса
{
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
}
hasDMA::~hasDMA()
{
   delete [] style;
}
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
   if (this == &hs)
      return *this;
   baseDMA::operator=(hs);     // копирование базовой части
   delete [] style;            // подготовка к операции new для style
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
   return *this;
}
std::ostream & operator<<(std::ostream & os, const hasDMA & hs)
{
   os << (const baseDMA &) hs;
   os << "Style: " << hs.style << std::endl;  // стиль
   return os;
}
```

Обратите внимание на новый момент в листингах 13.13 и 13.14: как производные классы могут использовать друзей базового класса. Вот, например, функция, дружественная классу `hasDMA`:

```
friend std::ostream & operator<<(std::ostream & os,
                                 const hasDMA & rs);
```

Поскольку эта функция дружественна классу `hasDMA`, она имеет доступ к члену `style`. Однако она не является дружественной классу `baseDMA`, и тогда как она может обращаться к членам `label` и `rating`? Для этого используется функция `operator<<()`, дружественная классу `baseDMA`. Есть еще одна проблема: поскольку друзья не являются функциями-членами, невозможно использовать разрешение контекста, чтобы указать, какую функцию следует вызвать. Для устранения этой проблемы можно использовать приведение типа, чтобы соответствующая функция была выбрана на основе сопоставления прототипов. Поэтому в коде выполняется приведение типа параметра `const hasDMA &` к типу аргумента `const baseDMA &`:

```
std::ostream & operator<<(std::ostream & os, const hasDMA & hs)
{
  // Приведение типа для соответствия operator<<(ostream & , const baseDMA &)
  os << (const baseDMA &) hs;
  os << "Стиль: " << hs.style << endl;
  return os;
}
```

Код в листинге 13.16 предназначен для проверки работы классов baseDMA, lacksDMA и hasDMA.

Листинг 13.16. usedma.cpp

```
// usedma.cpp -- наследование, друзья и динамическое выделение памяти
// Компилировать вместе с dma.cpp
#include <iostream>
#include "dma.h"
int main()
{
    using std::cout;
    using std::endl;
    baseDMA shirt("Portabelly", 8);
    lacksDMA balloon("red", "Blimpo", 4);
    hasDMA map("Mercator", "Buffalo Keys", 5);
    cout << "Displaying baseDMA object:\n";        // отображение объекта baseDMA
    cout << shirt << endl;
    cout << "Displaying lacksDMA object:\n";       // отображение объекта lacksDMA
    cout << balloon << endl;
    cout << "Displaying hasDMA object:\n";         // отображение объекта hasDMA
    cout << map << endl;
    lacksDMA balloon2(balloon);
    cout << "Result of lacksDMA copy:\n";          // результат копирования lacksDMA
    cout << balloon2 << endl;
    hasDMA map2;
    map2 = map;
    cout << "Result of hasDMA assignment:\n";      // результат присваивания hasDMA
    cout << map2 << endl;
    return 0;
}
```

Ниже показа вывод программы из листингов 13.14, 13.15 и 13.16:

```
Displaying baseDMA object:
Label: Portabelly
Rating: 8

Displaying lacksDMA object:
Label: Blimpo
Rating: 4
Color: red

Displaying hasDMA object:
Label: Buffalo Keys
Rating: 5
Style: Mercator

Result of lacksDMA copy:
Label: Blimpo
Rating: 4
Color: red

Result of hasDMA assignment:
Label: Buffalo Keys
Rating: 5
Style: Mercator
```

Обзор структуры класса

Язык C++ позволяет решать широкий круг задач программирования, и поэтому разработку класса невозможно свести к четким стандартным процедурам. Однако существуют некоторые часто применимые рекомендации, и сейчас самое время ознакомиться с ними, подведя итоги и выделив важные моменты.

Функции-члены, генерируемые компилятором

Как было отмечено в главе 12, компилятор автоматически генерирует ряд открытых функций-членов, которые называются *специальными функциями-членами*. Это название указывает на особую важность таких функций-членов. Рассмотрим некоторые из них еще раз.

Конструкторы по умолчанию

Конструктор по умолчанию либо не имеет аргументов вообще, либо для всех его аргументов предусмотрены значения по умолчанию. Если вы не определите ни одного конструктора, компилятор самостоятельно сгенерирует конструктор по умолчанию, иначе будет невозможно создавать объекты. Например, предположим, что имеется класс `Star`. Тогда для выполнения следующего кода необходим конструктор по умолчанию:

```
Star rigel;          // создание объекта без явной инициализации
Star pleiades[6];    // создание массива объектов
```

Кроме того, автоматический конструктор по умолчанию вызывает конструкторы по умолчанию для всех базовых классов и для всех членов, которые являются объектами другого класса.

Если в списке инициализаторов членов конструктора производного класса нет явного вызова конструктора базового класса, то компилятор использует конструктор по умолчанию базового класса для создания части базового класса в новом объекте. Если же в базовом классе конструктор по умолчанию отсутствует, то в этой ситуации появится сообщение об ошибке компиляции.

Если в классе определен хоть какой-нибудь конструктор, компилятор не генсрирует конструктор по умолчанию. В таком случае программист должен самостоятельно написать конструктор по умолчанию, если он необходим.

Помните, что одна из причин наличия конструкторов — необходимость правильной инициализации объектов. А если в классе есть указатели-члены, то они обязательно будут инициализированы. Поэтому рекомендуется иметь явный конструктор по умолчанию, который инициализирует все члены данных класса подходящими значениями.

Конструкторы копирования

Конструктор копирования для класса — это конструктор, который принимает в качестве аргумента объект типа этого класса. Как правило, такой параметр объявляется в виде константной ссылки на тип класса. Например, конструктор копирования для класса `Star` может иметь такой прототип:

```
Star(const Star &);
```

Конструктор копирования класса используется в следующих ситуациях:

- новый объект инициализируется объектом того же самого класса;
- объект передается в функцию по значению;
- функция возвращает объект по значению;
- компилятор генерирует временный объект.

Если в программе не используется конструктор копирования (явно или неявно), то компилятор предоставляет прототип, но не определение функции. Иначе программа определяет конструктор копирования, который выполняет почленную инициализацию – каждый член нового объекта инициализируется значением соответствующего члена исходного объекта. Если какой-то член сам является объектом класса, то почленная инициализация использует конструктор копирования, определенный для этого класса.

В некоторых случаях почленнная инициализация нежелательна. Например, для инициализации указателей-членов с помощью операции `new` обычно требуется глубокое копирование, как в примере класса `baseDMA`. Либо класс может содержать статическую переменную, которую нужно изменить. В подобных ситуациях необходимо определить собственный конструктор копирования.

Операции присваивания

Операция присваивания по умолчанию выполняет присваивание одного объекта другому объекту того же самого класса. Не путайте присваивание с инициализацией. Если оператор создает новый объект, то это инициализация, а если оператор изменяет значение существующего объекта, то это присваивание:

```
Star sirius;
Star alpha = sirius;     // инициализация
Star dogstar;
dogstar = sirius;        // присваивание
```

Присваивание по умолчанию выполняется почленно. Если какой-то член сам является объектом класса, то почленное присваивание по умолчанию использует операцию присваивания, определенную для этого класса. Если нужно явно определить конструктор копирования, то по тем же причинам должна быть также явно определена операция присваивания. Прототип для операции присваивания класса `Star` выглядит следующим образом:

```
Star & Star::operator=(const Star &);
```

Обратите внимание, что функция операции присваивания возвращает ссылку на объект `Star`. Типичный пример явной операции присваивания был продемонстрирован в классе `baseDMA`.

Компилятор не генерирует операций присваивания для присваивания одного типа другому. Предположим, что вам понадобилось присвоить строку объекту `Star`. Одним из способов является явное определение такой операции:

```
Star & Star::operator=(const char *) {...}
```

Другой способ – применение функции преобразования (см. ниже раздел "Соображения по поводу преобразований") для преобразования строки в объект `Star` с последующим использованием функции присваивания объекта `Star` объекту `Star`. Первый способ быстрее, но требует большего объема кода. Применение функции преобразования может привести к непонятным компилятору ситуациям.

Другие соображения относительно методов класса

При определении класса необходимо помнить о нескольких важных моментах. В последующих разделах описаны некоторые из них.

Соображения по поводу конструкторов

Конструкторы отличаются от других методов класса тем, что они создают новые объекты, а остальные методы вызываются существующими объектами. Это одна из причин, по которым конструкторы не наследуются. Наследование означает, что производный объект может использовать метод базового класса, а в случае конструкторов объект не существует до тех пор, пока конструктор не выполнит свою работу.

Соображения по поводу деструкторов

Не забывайте определить явный деструктор, который освобождает всю память, выделенную операцией `new` в конструкторах класса, и выполняет необходимую очистку того, что нужно, в объекте класса. Если класс будет использоваться в качестве базового, потребуется написать виртуальный деструктор, даже если класс не нуждается в деструкторе.

Соображения по поводу преобразований

Любой конструктор, который может быть вызван с ровно одним аргументом, определяет преобразование из типа аргумента в тип класса этого конструктора. Для примера рассмотрим следующие прототипы конструкторов для класса `Star`:

```
Star(const char *);                          // преобразует char * в Star
Star(const Spectral &, int members = 1);  // преобразует Spectral в Star
```

Конструкторы преобразования используются, скажем, когда преобразуемый тип передается функции, которая была определена как принимающая аргумент типа класса. Рассмотрим следующий код:

```
Star north;
north = "polaris";
```

Второй оператор вызывает функцию `Star::operator=(const Star &)`, используя конструктор `Star::Star(const char *)` для создания объекта `Star`, который передается в качестве аргумента в функцию операции присваивания. (Предполагается, что операция присваивания `(char *)` для `Star` не определена.)

Включение выражения `explicit` в прототип для конструктора с одним аргументом блокирует неявные преобразования, хотя и допускает явные:

```
class Star
{
   ...
public:
   explicit Star(const char *);
   ...
};
Star north;
north = "polaris";          // не разрешено
north = Star("polaris");    // разрешено
```

Для преобразования объекта класса в какой-то другой тип определяется функция преобразования (см. главу 11). Функция преобразования — это функция-член класса без аргументов или с объявленным возвращаемым типом, имя которой совпадает с типом, в который выполнятся преобразование. Несмотря на отсутствие объявленного возвращаемого типа, функция должна возвращать требуемое преобразованное значение.

Ниже показано несколько примеров:

```
Star::Star double() {...}          // преобразует star в double
Star::Star const char * () {...}   // преобразует в const char
```

Такие функции следует применять, только если они действительно нужны. В некоторых классах наличие функций преобразования повышает вероятность написания неоднозначного кода. Например, предположим, что определено преобразование типа `double` в тип `vector` из главы 11, и имеется следующий код:

```
vector ius(6.0, 0.0);
vector lux = ius + 20.2;           // неоднозначность
```

Компилятор может преобразовать `ius` в `double` и выполнить сложение чисел, либо преобразовать `20.2` в `vector` (используя один из конструкторов) и выполнить сложение векторов. На самом деле он ничего не сделает, а просто выдаст сообщение о неоднозначной конструкции.

C++11 позволяет использовать с функциями преобразования ключевое слово `explicit`. Как и в конструкторах, слово `explicit` разрешает явные преобразования с помощью приведения типа, но не разрешает неявные преобразования.

Передача объекта по значению и по ссылке

В общем случае, если вы создаете функцию с аргументом-объектом, ей нужно передавать объект по ссылке, а не по значению. Одной из причин этого является эффективность. Передача объекта по значению означает создание временной копии, а для этого необходим вызов конструктора копирования и последующий вызов деструктора. Эти вызовы занимают время, причем копирование большого объекта может длиться гораздо дольше, чем передача ссылки. Если функция не изменяет объект, ее аргумент следует объявить аргумент как `const`.

Еще одна причина передачи объектов по ссылке: в случае наследования с применением виртуальных функций функция, которая может принимать в качестве аргумента ссылку на базовый класс, может успешно работать и с производными классами, как было продемонстрировано выше в данной главе. (Этот вопрос также рассматривается в разделе "Соображения по поводу виртуальных методов" ниже в данной главе.)

Возврат объекта или возврат ссылки

Некоторые методы класса возвращают объекты. Вы, видимо, уже заметили, что некоторые члены возвращают непосредственно объекты, а другие возвращают ссылки. Иногда нужно, чтобы метод возвращал именно объект, но если это не обязательно, то вместо объекта лучше использовать ссылку. Рассмотрим этот момент более подробно.

Первое — единственное различие при кодировании между возвратом непосредственно объекта и возвратом ссылки заключается в прототипе функции и ее заголовке:

```
Star nova1(const Star &);          // возвращает объект Star
Star & nova2(const Star &);        // возвращает ссылку на Star
```

Вторая причина, по которой лучше возвращать ссылку, а не объект, состоит в том, что при возврате объекта создается временная копия возвращаемого объекта, и вызывающая программа получает доступ к этой копии. Поэтому возврат объекта означает потерю времени на вызов конструктора копирования для создания копии и на вызов деструктора для уничтожения этой копии. Возврат ссылки экономит и время, и память. Возврат объекта подобен передаче объекта по значению: оба эти процесса генерируют временные копии. Аналогично возврат ссылки похож на передачу объекта по ссылке: и вызывающая, и вызываемая функция работают с одним и тем же объектом.

Однако возврат ссылки возможен не всегда. Функция не может возвратить ссылку на временный объект, созданный функцией, ведь ссылка становится некорректной, когда функция завершает свою работу и объект исчезает. В этом случае код должен возвращать объект для создания копии, которая будет доступна вызывающей программе.

Если функция возвращает созданный в ней временный объект, то ссылку использовать не следует. Например, показанный ниже метод вызывает конструктор для создания нового объекта, и затем возвращает копию этого объекта:

```
Vector Vector::operator+(const Vector & b) const
{
   return Vector(x + b.x, y + b.y);
}
```

Если функция возвращает объект, переданный в нее через ссылку или указатель, то нужно возвращать объект по ссылке. Например, следующий код возвращает по ссылке либо объект, который вызывает функцию, либо объект, переданный в качестве аргумента:

```
const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
      return s;                  // объект-аргумент
   else
      return *this;              // вызывающий объект
}
```

Использование квалификатора `const`

По возможности старайтесь применять ключевое слово `const`. Оно позволяет гарантировать, что метод не изменит аргумент:

```
Star::Star(const char * s) {...} // не изменяет строку, на которую указывает s
```

Квалификатор `const` можно также применять для того, чтобы метод не модифицировал вызвавший его объект:

```
void Star::show() const {...}     // не изменяет вызывающий объект
```

Здесь `const` означает `const Star * this`, где `this` указывает на вызывающий объект.

Обычно функция, которая возвращает ссылку, может находиться в левой части оператора присваивания; это означает, что указываемому объекту можно присвоить значение. Но квалификатор `const` позволяет гарантировать, что возвращаемая ссылка или указатель не сможет использоваться для изменения данных в объекте:

```
const Stock & Stock::topval(const Stock & s) const
{
   if (s.total_val > total_val)
      return s;                  // объект-аргумент
   else
      return *this;              // вызывающий объект
}
```

Здесь метод возвращает ссылку либо на `this`, либо на `s`. Поскольку и `this`, и `s` объявлены как `const`, функция не может изменять их — а значит, и возвращаемая ссылка также должна быть объявлена как `const`.

Учтите, что если функция объявляет аргумент как ссылку или указатель на `const`, она не сможет передать этот аргумент в другую функцию, кроме случаев, когда эта другая функция также обещает не изменять аргумент.

Соображения по поводу открытого наследования

Понятно, что добавление наследования в программу увеличивает количество соображений. Рассмотрим некоторые из них.

Отношения является

Вы должны руководствоваться отношением *является*. Если производный класс не является разновидностью базового класса, то применять открытое наследование не стоит. Например, не следует порождать класс `Programmer` (Программист) от класса `Brain` (Мозг). Если вы хотите отразить свое глубокое убеждение, что у программистов есть мозги, нужно использовать объект класса `Brain` в качестве члена класса `Programmer`.

В некоторых случаях лучше всего создать абстрактный класс данных с чистыми виртуальными функциями и породить из него все нужные классы.

Помните, что одно из проявлений отношения *является* состоит в том, что указатель на базовый класс может указывать на объект производного класса, а ссылка на базовый класс может ссылаться на объект производного класса без явного приведения типа. И помните, что обратное неверно – т.е. указатель или ссылка на производный класс не могут ссылаться на объект базового класса без явного приведения типа. В зависимости от объявлений классов такое явное (нисходящее) приведение типа может иметь смысл, а может и не иметь. (Просмотрите еще раз рис. 13.4.)

Что не наследуется?

Конструкторы также не наследуются. То есть для создания производного объекта необходим вызов конструктора производного класса. Однако, как правило, конструкторы производного класса используют списки инициализаторов членов для вызова конструкторов базового класса, которые создают в производном объекте часть базового класса. Если конструктор производного класса не вызывает явно конструктор базового класса с помощью списка инициализаторов членов, то он использует конструктор по умолчанию базового класса. В цепочке наследования каждый класс может применять список инициализаторов членов для передачи информации обратно своему непосредственному базовому классу. В C++11 добавлен механизм, который делает возможным наследование конструкторов. Однако поведение по умолчанию по-прежнему не предусматривает наследование конструкторов.

Деструкторы также не наследуются. Но при уничтожении объекта программа сначала вызывает деструктор производного класса, а затем деструктор базового класса. Если в базовом классе используется деструктор по умолчанию, то компилятор генерирует деструктор по умолчанию производного класса. В общем случае, если класс используется как базовый, его деструктор должен быть виртуальным.

Соображения по поводу операции присваивания

Операции присваивания не наследуются. Причина очень проста. Унаследованный метод должен иметь такую же сигнатуру функции в производном классе, что и в базовом. Однако сигнатура операции присваивания изменяется от класса к классу, т.к. ее формальный параметр совпадает с типом класса. Операции присваивания обладают некоторыми интересными свойствами, которые будут рассмотрены ниже.

Если компилятор обнаруживает, что программа присваивает один объект другому объекту того же класса, он автоматически снабжает этот класс операцией присваивания. Версия по умолчанию или неявная версия этой операции использует почленное присваивание, когда в каждый член целевого объекта копируется значение соответ-

ствующего члена исходного объекта. Однако если объект принадлежит производному классу, то для выполнения присваивания части базового класса в объекте производного класса компилятор применяет операцию присваивания базового класса. Если для базового класса явно указана операция присваивания, то используется она. Аналогично, если класс содержит член, являющийся объектом другого класса, то для этого члена применяется операция присваивания собственного класса.

Вы уже неоднократно видели, что если конструкторы класса применяют операцию new для инициализации указателей, то необходимо предусмотреть явную операцию присваивания. Поскольку в C++ для базовой части производных объектов используется операция присваивания из базового класса, не нужно переопределять операцию присваивания для производного класса — *за исключением* тех случаев, когда добавляются члены данных, которые требуют особой осторожности. Например, в классе baseDMA присваивание определяется явно, но производный класс lacksDMA использует неявную операцию присваивания, сгенерированную для данного класса.

Предположим, однако, что производный класс применяет операцию new, и поэтому необходимо написать явную операцию присваивания. Операция должна работать для каждого члена класса, а не только для новых членов. Как это можно сделать, продемонстрировано в классе hasDMA:

```
hasDMA & hasDMA::operator=(const hasDMA & hs)
{
   if (this == &hs)
      return *this;
   baseDMA::operator=(hs);   // копирование базовой части
   delete [] style;          // подготовка к операции new для style
   style = new char[std::strlen(hs.style) + 1];
   std::strcpy(style, hs.style);
   return *this;
}
```

А как насчет присваивания объекта производного класса объекту базового класса? (Учтите, что это не то же самое, что инициализация ссылки на базовый класс объектом производного класса.) Рассмотрим следующий пример:

```
Brass blips;                                          // базовый класс
BrassPlus snips("Rafe Plosh", 91191,3993.19, 600.0, 0.12); // производный класс
blips = snips;          // присваивание производного объекта базовому объекту
```

Какая из операций присваивания используется? Вспомните, что оператор присваивания транслируется в метод, который вызывается объектом, расположенным в левой части:

```
blips.operator=(snips);
```

Здесь слева находится объект Brass, поэтому он вызывает функцию Brass::operator=(const Brass &). Отношение *является* позволяет ссылке на Brass ссылаться на объект производного класса, такой как snips. Операция присваивания имеет дело только с членами базового класса, поэтому член maxLoan и остальные члены из класса BrassPlus объекта snips в присваивании игнорируются. То есть производный объект можно присвоить базовому, но при этом задействуются только члены базового класса.

А можно ли, наоборот, присвоить объект базового класса объекту производного класса? Давайте рассмотрим пример:

```
Brass gp("Griff Hexbait", 21234, 1200);  // базовый класс
BrassPlus temp;                          // производный класс
temp = gp;                               // возможно ли это?
```

Здесь оператор присваивания транслируется в конструкцию

```
temp.operator=(gp);
```

Слева указан объект `BrassPlus`, поэтому он вызывает функцию `BrassPlus:: operator=(const BrassPlus &)`. Однако ссылка на производный класс не может автоматически указывать на объект базового класса, поэтому данный код *не будет* работать, если нет соответствующего конструктора преобразования:

```
BrassPlus(const Brass &);
```

Может оказаться, как в случае класса `BrassPlus`, что конструктор преобразования содержит базовые аргументы и дополнительные аргументы, причем для дополнительных аргументов указаны значения по умолчанию:

```
BrassPlus(const Brass & ba, double ml = 500, double r = 0.1);
```

При наличии конструктора преобразования программа использует его для создания из объекта `gp` временного объекта `BrassPlus`, который затем передается в качестве аргумента операции присваивания.

Но можно поступить и по-другому — определить операцию присваивания базового класса производному классу:

```
BrassPlus & BrassPlus ::operator=(const Brass &) {...}
```

Здесь типы в точности соответствуют оператору присваивания, поэтому преобразования типа не нужны.

В общем, на вопрос "Можно ли присвоить объект базового класса производному объекту?" следует ответить: "Возможно". Можно, если производный класс имеет конструктор, который определяет преобразование объекта базового класса в объект производного класса. Можно и тогда, когда в производном классе определена операция присваивания объекта базового класса объекту производного класса. Если ни того, ни другого нет, присваивание невозможно без явного приведения типа.

Закрытые и защищенные члены

Помните, что защищенные члены ведут себя как открытые члены для производного класса и как закрытые члены для внешнего мира. Производный класс может напрямую обращаться к защищенным членам базового класса, однако доступ к закрытым членам возможен только через методы базового класса. Значит, объявление членов базового класса закрытыми усиливает защиту, а объявление их защищенными упрощает кодирование и ускоряет доступ. Страуструп в одной из своих книг указывает, что лучше применять закрытые члены данных, чем защищенные, однако защищенные методы тоже нужны.

Соображения по поводу виртуальных методов

При разработке базового класса необходимо решить, делать ли методы класса виртуальными. Если метод потребуется переопределять в производном классе, то в базовом классе его следует определить как виртуальный. Тогда будет выполняться позднее, или динамическое, связывание. Если метод не нужно переопределять, то не стоит делать его виртуальным. Это не помешает кому-либо переопределить метод, однако продемонстрирует, что вы не хотите его переопределять. Учтите, что некорректный код может обойти динамическое связывание.

Рассмотрим, к примеру, две следующих функции:

```
void show(const Brass & rba)
{
```

```
    rba.ViewAcct();
    cout << endl;
}
void inadequate(Brass ba)
{
    ba.ViewAcct();
    cout << endl;
}
```

Первая функция передает объект по ссылке, а вторая — по значению.

Теперь предположим, что каждая из этих функций вызывается с аргументом производного класса:

```
BrassPlus buzz("Buzz Parsec", 00001111, 4300);
show(buzz);
inadequate(buzz);
```

В вызове функции `show()` аргумент `rba` является ссылкой на объект `buzz` типа `BrassPlus`, поэтому `rba.ViewAcct()` интерпретируется как версия `BrassPlus`, что и должно быть. Но в функции `inadequate()`, которая передает объект по значению, `ba` является объектом `Brass`, созданным конструктором `Brass(const Brass &)`. (Автоматическое восходящее приведение позволяет аргументу конструктора ссылаться на объект `BrassPlus`.) Поэтому в `inadequate()` вызов `ba.ViewAcct()` считается версией `Brass`, и выводится только компонент `Brass` объекта `buzz`.

Соображения по поводу деструкторов

Как уже было неоднократно сказано, деструктор базового класса должен быть виртуальным. Тогда при удалении производного объекта через указатель или ссылку базового класса на объект программа использует деструктор производного класса, и затем деструктор базового класса, а не просто деструктор базового класса.

Соображения по поводу дружественных функций

Дружественная функция фактически не является членом класса и поэтому не наследуется. Однако может понадобиться, чтобы друг производного класса использовал дружественную функцию базового класса. Для этого необходимо привести тип ссылки или указателя на производный класс к эквиваленту базового класса, а затем вызвать дружественную функцию базового класса с помощью полученного указателя или ссылки:

```
ostream & operator<<(ostream & os, const hasDMA & hs)
{
  // Приведение типа для соответствия operator<<(ostream & , const baseDMA &)
  os << (const baseDMA &) hs;
  os << "Style: " << hs.style << endl;
  return os;
}
```

Для приведения типа можно также использовать операцию `dynamic_cast<>`, которая рассматривается в главе 15:

```
os << dynamic_cast<const baseDMA &> (hs);
```

По причинам, изложенным в главе 15, эта форма приведения типа является наиболее предпочтительной.

Соображения по поводу использования методов базового класса

Открытые производные объекты могут использовать методы базового класса многими способами.

- Производный объект автоматически использует унаследованные методы базового класса, если в производном классе эти методы не переопределены.
- Деструктор производного класса автоматически вызывает конструктор базового класса.
- Конструктор производного класса автоматически вызывает конструктор по умолчанию базового класса, если в списке инициализаторов членов не указан другой конструктор.
- Конструктор производного класса явно вызывает конструктор базового класса, указанный в списке инициализаторов членов.
- Методы производного класса могут применять операцию разрешения контекста для вызова открытых и защищенных методов базового класса.
- Друзья производного класса могут приводить тип ссылки или указателя на производный класс к ссылке или указателю на базовый класс и затем использовать данную ссылку или указатель для вызова дружественной функции базового класса.

Сводка функций классов

Функции классов C++ имеют множество разновидностей. Некоторые могут наследоваться, а другие нет. Некоторые функции могут быть как функциями-членами, так и дружественными, а другие — только функциями-членами. В табл. 13.1 приведена сводка по этим свойствам. В ней запись *op*= означает операции присваивания вида +=, *= и т.д. Обратите внимание, что свойства операций *op*= не отличаются от свойств категории "Другие операции". Операция *op*= вынесена отдельно, чтобы подчеркнуть, что эти операции ведут себя не так, как операция =.

Таблица 13.1. Свойства функций-членов

Функция	Наследуется	Член или друг	Генерируется по умолчанию	Может быть виртуальной	Может иметь возвращаемый тип
Конструктор	Нет	Член	Да	Нет	Нет
Деструктор	Нет	Член	Да	Да	Нет
=	Нет	Член	Да	Да	Да
&	Да	Оба	Да	Да	Да
Преобразование	Да	Член	Нет	Да	Нет
()	Да	Член	Нет	Да	Да
[]	Да	Член	Нет	Да	Да
->	Да	Член	Нет	Да	Да
op=	Да	Оба	Нет	Да	Да
new	Да	Статический член	Нет	Нет	void *
delete	Да	Статический член	Нет	Нет	void
Другие операции	Да	Оба	Нет	Да	Да
Другие члены	Да	Член	Нет	Да	Да
Друзья	Нет	Друг	Нет	Нет	Да

Резюме

Наследование позволяет адаптировать программный код к конкретным потребностям с помощью определения нового (производного) класса из существующего (базового). Открытое наследование моделирует отношение *является*, а это означает, что объект производного класса должен быть разновидностью объекта базового класса. Как часть модели *является*, производный класс наследует члены данных и большинство методов базового класса. Однако производный класс не наследует конструкторы, деструкторы и операции присваивания базового класса. Производный класс может обращаться к открытым и защищенным членам базового класса непосредственно, а к закрытым членам базового класса — через открытые и защищенные методы базового класса. Затем в класс можно добавлять новые члены данных и методы, а также использовать производный класс в качестве базового для дальнейшей разработки.

В каждом производном классе должен быть собственный конструктор. Когда программа создает объект производного класса, она сначала вызывает конструктор базового класса, а затем конструктор производного класса. При удалении объекта программа сначала вызывает деструктор производного класса, а затем деструктор базового класса.

Если класс предполагается использовать в качестве базового, то можно использовать защищенные члены, а не закрытые — тогда производные классы будут иметь прямой доступ к данным-членам. Однако применение закрытых членов обычно снижает вероятность появления программных ошибок. Если в производном классе планируется переопределение какого-то метода базового класса, его необходимо сделать виртуальной функцией, объявив его с ключевым словом `virtual`. Это позволяет управлять объектами, на которые указывают указатели или ссылки, на основе типа объекта, а не на основе типа ссылки или указателя. В частности, должен быть виртуальным деструктор для базового класса.

Возможно, понадобится определить абстрактный базовый класс, который определяет интерфейс без деталей реализации. Например, можно определить абстрактный класс `Shape` (Фигура), а от него порождать отдельные классы фигур, такие как `Circle` (Круг) и `Square` (Прямоугольник). Абстрактный базовый класс должен содержать хотя бы один чистый виртуальный метод. Для объявления чистой виртуальной функции нужно в объявлении после закрывающей скобки добавить конструкцию `= 0`:

```
virtual double area() const = 0;
```

Определять чистые виртуальные методы не нужно; кроме того, невозможно создать объект класса, который содержит чистые виртуальные члены. Чистые виртуальные функции служат только для определения общего интерфейса, который будет использоваться производными классами.

Вопросы для самоконтроля

1. Что производный класс наследует от базового класса?
2. Что производный класс не наследует от базового класса?
3. Предположим, что возвращаемый тип для функции `baseDMA::operator=()` определен как `void`, а не `baseDMA &`. Как это повлияет на программу? Что случится, если возвращаемым типом будет `baseDMA`, а не `baseDMA &`?
4. В каком порядке вызываются конструкторы и деструкторы класса при создании и удалении объекта производного класса?

5. Если производный класс не добавляет члены данных в базовый класс, то нужны ли конструкторы для производного класса?

6. Предположим, что и в базовом, и в производном классе определен метод с одним и тем же именем, и производный класс вызывает этот метод. Который метод будет вызван?

7. В каких случаях производный класс должен определять операцию присваивания?

8. Можно ли присвоить адрес объекта производного класса указателю на базовый класс? Можно ли присвоить адрес объекта базового класса указателю на производный класс?

9. Можно ли присвоить объект производного класса объекту базового класса? Можно ли присвоить объект базового класса объекту производного класса?

10. Предположим, что определена функция, которая принимает в качестве аргумента ссылку на объект базового класса. Почему эта функция может также использовать в качестве аргумента объект производного класса?

11. Предположим, что определена функция, которая принимает в качестве аргумента объект базового класса (т.е. функция передает объект базового класса по значению). Почему эта функция может также использовать в качестве аргумента объект производного класса?

12. Почему обычно лучше передавать объекты по ссылке, а не по значению?

13. Предположим, что `Corporation` — базовый класс, а `PublicCorporation` — производный. Допустим также, что в каждом из этих классов определен метод `head()`, `ph` является указателем на тип `Corporation`, а переменной `ph` присвоен адрес объекта `PublicCorporation`. Как интерпретируется `ph->head()`, если в базовом классе метод `head()` определен как:

    а. обычный не виртуальный метод;

    б. виртуальный метод.

14. Есть ли ошибки в следующем коде, и если есть, то какие?

```
class Kitchen
{
private:
   double kit_sq_ft;
public:
   Kitchen() { kit_sq_ft = 0.0; }
   virtual double area() const { return kit_sq_ft * kit_sq_ft; }
};
class House : public Kitchen
{
private:
   double all_sq_ft;
public:
   House() { all_sq_ft += kit_sq_ft; }
   double area(const char *s) const { cout << s; return all_sq_ft; }
};
```

Упражнения по программированию

1. Начните со следующего объявления класса:

```
// Базовый класс
class Cd {                    // представляет компакт-диск
private:
   char performers[50];
   char label[20];
   int selections;            // количество сборников
   double playtime;           // время воспроизведения в минутах
public:
   Cd(char * s1, char * s2, int n, double x);
   Cd(const Cd & d);
   Cd();
   ~Cd();
   void Report() const;       // выводит все данные о компакт-диске
   Cd & operator=(const Cd & d);
};
```

Породите класс `Classic`, добавив массив членов `char`, которые будут хранить строку с названием основного произведения на компакт-диске. Если необходимо, чтобы какие-то функции в базовом классе были виртуальными, измените объявление базового класса. Если объявленный метод не нужен, удалите его из определения. Протестируйте результат с помощью следующей программы:

```
#include <iostream>
using namespace std;
#include "classic.h"          // будет содержать #include cd.h
void Bravo(const Cd & disk);
int main()
{
   Cd c1("Beatles", "Capitol", 14, 35.5);
   Classic c2 = Classic("Piano Sonata in B flat, Fantasia in C",
                        "Alfred Brendel", "Philips", 2, 57.17);
   Cd *pcd = &c1;
   // Непосредственное использование объектов
   cout << "Using object directly:\n";
   c1.Report();               // использование метода Cd
   c2.Report();               // использование метода Classic
   // Использование указателя на объекты типа cd *
   cout << "Using type cd * pointer to objects:\n";
   pcd->Report();             // использование метода Cd для объекта cd
   pcd = &c2;
   pcd->Report();             // использование метода Classic для объекта classic
   // Вызов функции с аргументом-ссылкой на Cd
   cout << "Calling a function with a Cd reference argument:\n";
   Bravo(c1);
   Bravo(c2);
   // Тестирование присваивания
   cout << "Testing assignment: ";
   Classic copy;
   copy = c2;
   copy.Report()
   return 0;
}
void Bravo(const Cd & disk)
{
   disk.Report();
}
```

2. Выполните упражнение 1, но для различных строк, используемых двумя классами, вместо массивов фиксированного размера применяйте динамическое выделение памяти.

3. Перепишите иерархию классов baseDMA-lacksDMA-hasDMA таким образом, чтобы все три класса были порождены от абстрактного базового класса. Протестируйте результат с помощью программы, подобной приведенной в листинге 13.10. То есть она должна использовать массив указателей на абстрактный базовый класс и позволять пользователю принимать во время работы программы решения о том, объекты какого типа создавать. Добавьте в определения классов виртуальные методы View() для управления выводом данных.

4. Союз программистов-меценатов собирает коллекцию бутылочного портвейна. Для ее описания администратор союза разработал класс Port:

```
#include <iostream>
using namespace std;
class Port                          // портвейн
{
private:
   char * brand;
   char style[20];                  // например, tawny (золотистый),
                                    // ruby (рубиновый), vintage (марочный)
   int bottles;
public:
   Port(const char * br = "none", const char * st = "none", int b = 0);
   Port(const Port & p);            // конструктор копирования
   virtual ~Port() { delete [] brand; }
   Port & operator=(const Port & p);
   Port & operator+=(int b);        // добавляет b к bottles
   Port & operator-=(int b);        // вычитает b из bottles, если это возможно
   int BottleCount() const { return bottles; }
   virtual void Show() const;
   friend ostream & operator<<(ostream & os, const Port & p);
};
```

Метод Show() выводит информацию в следующем формате:

```
Brand: Gallo
Kind: tawny
Bottles: 20
```

Функция operator<<() представляет информацию в следующем формате (без символа новой строки в конце):

```
Gallo, tawny, 20
```

Завершив определения методов для класса Port, администратор написал производный класс VintagePort, прежде чем был уволен:

```
class VintagePort : public Port     // style обязательно = "vintage"
{
private:
   char * nickname;                 // т.е. "The Noble", "Old Velvet" и т.д.
   int year;                        // год сбора
public:
   VintagePort();
   VintagePort(const char * br, int b, const char * nn, int y);
   VintagePort(const VintagePort & vp);
```

```
    ~VintagePort() { delete [] nickname; }
    VintagePort & operator=(const VintagePort & vp);
    void Show() const;
    friend ostream & operator<<(ostream & os, const VintagePort & vp);
};
```

Вам поручено завершить разработку класса `VintagePort`.

а. Первое задание — нужно заново создать определения методов `Port`, т.к. предыдущий администратор уничтожил свой код.

б. Второе задание — объясните, почему одни методы переопределены, а другие нет.

в. Третье задание — объясните, почему функции `operator=()` и `operator<<()` не определены как виртуальные.

г. Четвертое задание — обеспечьте определения для методов `VintagePort`.

14

Повторное использование кода в C++

В ЭТОЙ ГЛАВЕ...

- Отношения *содержит*
- Классы с объектами-членами (включение)
- Класс шаблона `valarray`
- Закрытое и защищенное наследование
- Множественное наследование
- Виртуальные базовые классы
- Создание шаблонов классов
- Использование шаблонов классов
- Специализации шаблонов

Возможность повторного использования кода входит в число основных задач языка C++. Один из механизмов достижения этой цели — открытое наследование, но это не единственный механизм. В этой главе будут рассмотрены другие механизмы. Одним из приемов является использование членов класса, которые сами представляют собой объекты другого класса. Это называется *включением (containment)*, или *композицией (*composition*)*, или *иерархическим представлением (layering)*. Кроме того, можно применять закрытое или защищенное наследование. Включение, закрытое наследование и защищенное наследование обычно используются для создания отношений *содержит* – когда вновь создаваемый класс содержит в себе объект другого, уже существующего класса. Например, класс `HomeTheater` (Домашний кинотеатр) может содержать объект `BluRayPlayer` (DVD-плеер). Множественное наследование позволяет создавать классы, которые наследуются от двух или более базовых классов и обладают их совокупной функциональностью.

В главе 10 были рассмотрены шаблоны функций. В этой главе описываются шаблоны классов — еще один способ повторного использования кода. Шаблон класса позволяет определить общие свойства класса. Затем этот шаблон можно использовать для создания конкретных классов, предназначенных для специфических целей. Например, можно создать общий шаблон стека, а затем применить его для создания класса, представляющего стек значений типа `int`, и другого класса, представляющего стек значений типа `double`. Возможно даже создание класса, представляющего стек стеков.

Классы с объектами-членами

Начнем с классов, которые содержат в качестве членов объекты других классов. Некоторые классы — например, класс `string` или стандартные шаблоны классов C++, рассматриваемые в главе 16 — предоставляют хорошую основу для создания более сложных классов. Ниже приведен конкретный пример.

Кто такой студент? Тот, кто поступил в учебное заведение? Тот, кто занимается исследовательской работой? Беглец от трудностей реального мира? Кто-то с именем и набором оценок? Ясно, что последнее определение — совершенно неадекватная характеристика студента, однако она вполне годится для простого компьютерного представления. И сейчас мы создадим класс `Student`, основанный на этом определении.

Если свести представление студента к имени и набору оценок, то можно использовать класс, содержащий два члена — один для представления имени и другой для набора оценок. Для имени можно использовать символьный массив, но он налагает ограничения на длину имени. Или же можно использовать указатель на `char` и динамическое выделение памяти. Но как показано в главе 12, для этого нужен большой объем дополнительного кода. Лучше воспользоваться объектом существующего класса, для которого кто-то уже проделал всю необходимую работу. Например, можно выбрать объект класса `String` (см. главу 12) или стандартный класс `string` из C++. Проще выбрать класс `string`, т.к. библиотека C++ уже содержит весь необходимый код, а также отличную реализацию. (Для использования класса `String` потребуется включить в проект файл `string1.cpp`.)

Аналогично можно представить и набор оценок. Использование массива фиксированной длины ограничивает количество оценок. Динамическое выделение памяти увеличивает размер исходного кода. Можно разработать собственный класс, используя динамическое распределение памяти для создания массива, а можно воспользоваться подходящим классом из стандартной библиотеки C++.

Разработка собственного класса не представляет особой трудности. Это не сложно, поскольку массив double имеет много общего с массивом char. Поэтому при создании класса массива из double можно использовать принципы построения класса String. В действительности так и было сделано в предыдущих изданиях этой книги.

Конечно, еще проще, если библиотека уже содержит подходящий класс. И такой класс существует — это valarray.

Класс valarray: краткий обзор

Класс valarray поддерживается заголовочным файлом valaray и предназначен для работы с числовыми значениями (или с классами с аналогичными свойствами). Он поддерживает операции суммирования содержимого массива и поиск максимального и минимального значений в массиве. Поскольку класс valarray может работать с данными различных типов, он определен как шаблонный класс. Позднее вы научитесь создавать шаблонные классы, а пока просто достаточно знать, как его использовать.

При объявлении объекта на основе шаблона необходимо указать его конкретный тип. Поэтому за идентификатором valarray должны следовать угловые скобки, содержащие требуемый тип:

```
valarray<int> q_values;       // массив значений int
valarray<double> weights;     // массив значений double
```

Вы уже знакомы с этим несложным синтаксисом: в главе 4 подобным образом определялись классы vector и array. (Эти классы тоже могут содержать числа, но они не обеспечивают такой объем арифметической поддержки, как класс valarray.)

Поскольку объекты valarray представляют собой экземпляры классов, необходимо иметь представление о конструкторах и других методах класса. Вот несколько примеров, использующих различные конструкторы:

```
double gpa[5] = {3.1, 3.5, 3.8, 2.9, 3.3};
valarray<double> v1;          // массив элементов double, размер 0
valarray<int> v2(8);          // массив из 8 элементов int
valarray<int> v3(10,8);       // массив из 8 элементов int, и каждый равен 10
valarray<double> v4(gpa, 4);  // массив из 4 элементов, равных
                              // первым 4 элементам массива gpa
```

В примерах видно, что можно создать пустой массив нулевого размера, пустой массив заданного размера, массив, всем элементам которого присвоены одинаковые значения, и массив, инициализированный значениями из обычного массива. Кроме того, в C++11 можно применять списки инициализаторов:

```
valarray<int> v5 = {20, 32, 17, 9};  // C++11
```

Ниже описаны некоторые методы класса valarray:

- operator[]() — обеспечивает доступ к отдельным элементам;
- size() — возвращает количество элементов;
- sum() — возвращает сумму значений элементов;
- max() — возвращает максимальный элемент;
- min() — возвращает минимальный элемент.

В этом классе доступно еще много методов (часть из них описана в главе 16), но вы уже знаете достаточно, чтобы перейти к рассмотрению примера.

Проект класса Student

В классе Student планируется использовать объект string для представления имени и объект valarray<double> для хранения набора оценок. Как это сделать? Может возникнуть желание породить класс Student от этих двух классов. Это было бы примером множественного открытого наследования, возможного в C++, но в данном случае это неприемлемо. Дело в том, что отношение класса Student с этими классами не соответствует модели *является*. Студент – это не имя и не массив оценок. Здесь мы имеем дело с отношением *содержит*. У студента есть имя, и у студента есть набор оценок. Обычно для моделирования отношений *содержит* в C++ используется композиция или включение, когда класс содержит члены, являющиеся объектами других классов. Например, можно начать объявление класса Student следующим образом:

```
class Student
{
private:
  string name;              // используется объект string для имени
  valarray<double> scores; // используется объект valarray<double> для оценок
    ...
};
```

Как обычно, данные-члены класса сделаны закрытыми. Это значит, что функции-члены класса Student могут использовать открытые интерфейсы классов string и valarray<double> для доступа к объектам name и scores, но доступ извне к этим объектам невозможен. Внешний мир может обращаться к string и valarray<double> только через открытый интерфейс класса Student (рис. 14.1). Обычно в такой ситуации говорят, что класс Student содержит реализацию своих объектов-членов, но не наследует их интерфейс. Например, объект Student использует для представления имени реализацию string, а не char * name или char name[26]. Однако объект Student не может изначально использовать функцию string operator+=() для добавления символов.

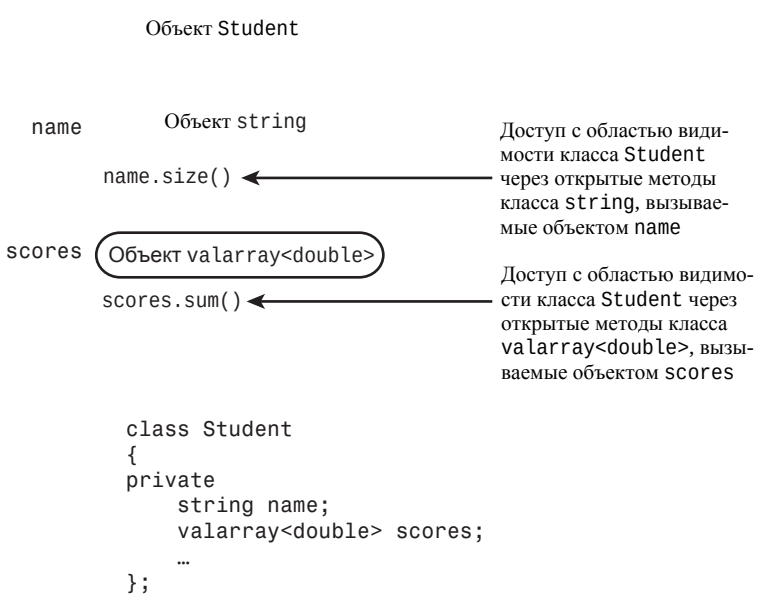


***Рис. 14.1.** Объекты внутри объектов: включение*

Интерфейсы и реализации

При открытом наследовании класс наследует интерфейс и, возможно, реализацию. (Чистые виртуальные функции базового класса могут предоставлять интерфейс без реализации.) Наличие интерфейса характерно для отношения *является*. А при композиции класс имеет реализацию без интерфейса. Отсутствие наследования интерфейса характерно для отношения *содержит*.

То, что объект класса автоматически не получает интерфейс включаемого объекта, полезно для отношения *содержит*. Например, класс `string` перегружает операцию + для конкатенации двух строк, но выполнять конкатенацию двух объектов `Student` бессмысленно. Поэтому в данном случае не имеет смысла использовать открытое наследование. Правда, часть интерфейса наследуемого класса может пригодиться и в новом классе. Например, можно использовать метод `operator<()` из интерфейса класса `string` для сортировки объектов `Student` по имени. Для этого потребуется определить функцию-член `Student::operator<()`, внутри которой вызывается функция `string::operator<()`. Рассмотрим это подробнее.

Пример класса Student

Нам нужно определение класса `Student`. Оно, конечно же, должно содержать конструкторы и, как минимум, несколько функций, реализующих интерфейс для класса `Student`. Все это показано в листинге 14.1, определяющем встроенные конструкторы и несколько дружественных функций для ввода и вывода.

Листинг 14.1. studentc.h

```
// studentc.h -- определение класса Student с использованием включения
#ifndef STUDENTC_H_
#define STUDENTC_H_

#include <iostream>
#include <string>
#include <valarray>

class Student
{
private:
    typedef std::valarray<double> ArrayDb;
    std::string name;      // включенный объект
    ArrayDb scores;        // включенный объект

    // Закрытый метод для вывода оценок
    std::ostream & arr_out(std::ostream & os) const;
public:
    Student() : name("Null Student"), scores() {}
    explicit Student(const std::string & s)
        : name(s), scores() {}
    explicit Student(int n) : name("Nully"), scores(n) {}
    Student(const std::string & s, int n)
        : name(s), scores(n) {}
    Student(const std::string & s, const ArrayDb & a)
        : name(s), scores(a) {}
    Student(const char * str, const double * pd, int n)
        : name(str), scores(pd, n) {}
    ~Student() {}
    double Average() const;
    const std::string & Name() const;
```

```
    double & operator[](int i);
    double operator[](int i) const;

    // Друзья
    // Ввод
    friend std::istream & operator>>(std::istream & is,
              Student & stu);  // 1 слово
    friend std::istream & getline(std::istream & is,
              Student & stu);  // 1 строка
    // Вывод
    friend std::ostream & operator<<(std::ostream & os,
       const Student & stu);
};
#endif
```

С целью упрощения класс `Student` содержит следующее определение `typedef`:

```
typedef std::valarray<double> ArrayDb;
```

Это позволяет в остальном коде использовать вместо `std::valarray<double>` более удобную обозначение `ArrayDb`. Теперь методы и друзья класса могут ссылаться на тип `ArrayDb`. Размещение объявления `typedef` в закрытом разделе определения класса означает, что его можно применять только внутри реализации класса `Student`, однако оно недоступно внешним пользователям класса.

Обратите внимание на использование ключевого слова `explicit`:

```
explicit Student(const std::string & s)
    : name(s), scores() {}
explicit Student(int n) : name("Nully"), scores(n) {}
```

Вспомните, что конструктор, который можно вызвать с одним аргументом, работает как функция неявного преобразования типа аргумента в тип класса. Часто такое поведение нежелательно. Например, во втором конструкторе первый аргумент представляет количество элементов в массиве, а не значение для массива, и выполняемое конструктором преобразование `int` в `Student` не имеет смысла. Указание ключевого слова `explicit` отключает неявное преобразование. Без него станет возможным следующий код:

```
Student doh("Homer", 10); // сохраняет "Homer", создает массив из 10 элементов
doh = 5;        // сбрасывает имя в "Nully", а массив — в пустой из 5 элементов
```

Здесь невнимательный программист вместо `doh[0]` напечатал `doh`. Без ключевого слова `explicit` в конструкторе значение `5` будет преобразовано во временный объект `Student` с помощью конструктора `Student(5)`, и члену `name` будет присвоено значение `"Nully"`. Затем операция присваивания заменит первоначальный `doh` содержимым этого временного объекта. При наличии ключевого слова `explicit` компилятор будет считать такую операцию присваивания ошибочной.

> **Язык C++ и ограничения**
>
> В C++ имеется много средств, позволяющих программисту накладывать определенные ограничения на программные конструкции: `explicit` — отключение неявного преобразования в конструкторах с одним аргументом, `const` — ограничение применения методов преобразования данных и т.д. Причина проста: лучше получать ошибки времени компиляции, чем ошибки времени выполнения.

Инициализация включенных объектов

Обратите внимание, что для инициализации объектов-членов `name` и `scores` все конструкторы используют уже хорошо известный синтаксис списка инициализаторов членов. Но раньше в этой книге он применялся для инициализации членов встроенных типов, например:

```
Queue::Queue(int qs) : qsize(qs) {...}  // инициализация qsize значением qs
```

В этом коде в списке инициализаторов членов указано имя члена (`qsize`). А в некоторых примерах конструкторы использовали список инициализаторов членов для инициализации порции производного объекта, которая взята из базового класса, например:

```
hasDMA::hasDMA(const hasDMA & hs) : baseDMA(hs) {...}
```

Для *унаследованных* объектов конструкторы используют в списке инициализаторов членов имя *класса*, чтобы вызвать конкретный конструктор базового класса. Для объектов-*членов* конструкторы используют имя *члена*. Взгляните, например, на последний конструктор из листинга 14.1:

```
Student(const char * str, const double * pd, int n)
    : name(str), scores(pd, n) {}
```

Поскольку этот конструктор инициализирует объекты-члены, а не унаследованные объекты, в списке инициализаторов он использует имена членов, а не классов. Каждый элемент в списке инициализации вызывает соответствующий конструктор. Так, элемент `name(str)` вызывает конструктор `string(const char *)`, а элемент `scores(pd, n)` вызывает конструктор `ArrayDb(const double *, int)`, который в силу определения `typedef` на самом деле является таким конструктором:

```
valarray<double>(const double *, int)
```

Что произойдет, если не использовать список инициализаторов? В C++ все объекты-члены унаследованных компонентов должны быть созданы до того, как будут созданы все остальные объекты. Значит, без списка инициализаторов C++ использует конструктор по умолчанию, определенный для классов объектов-членов.

Порядок инициализации

При наличии более одного объекта в списке инициализаторов эти объекты инициализируются в том порядке, в котором они объявлены, а не в порядке, в котором они содержатся в списке инициализаторов. Например, предположим, что конструктор `Student` имеет следующий вид:

```
student(const char * str, const double * pd, int n)
     : scores(pd, n), name(str) {}
```

Член `name` будет инициализирован первым, поскольку он объявлен первым в определении класса. В данном случае точный порядок инициализации не важен, однако он будет существенным, если в коде значение одного члена используется в составе выражения для инициализации другого члена.

Использование интерфейса для включенного объекта

Интерфейс для включенных объектов не является открытым, но его можно использовать внутри методов класса.

Например, вот как определить функцию, возвращающую среднее значение оценок студента:

```
double Student::Average() const
{
   if (scores.size() > 0)
      return scores.sum()/scores.size();
   else
      return 0;
}
```

Эта функция определяет метод, который может быть вызван объектом `Student`. Внутри этого метода применяются методы `sum()` и `size()`. Поскольку `scores` является объектом `valarray`, он может вызывать функции-члены класса `valarray`. В общем, объект `Student` вызывает метод `Student`, а тот использует включенный объект `valarray`, чтобы вызывать методы `valarray`.

Аналогично можно определить дружественную функцию, которая пользуется версией операции << из класса `string`:

```
// Использование версии операции << из класса string
ostream & operator<<(ostream & os, const Student & stu)
{
   os << "Scores for " << stu.name << ":\n";
   ...
}
```

Поскольку член `stu.name` является объектом `string`, он вызывает функцию `operator<<(ostream &, const string&)`, являющуюся частью пакета класса `string`. Обратите внимание, что функция `operator<<(ostream & os, const Student & stu)` должна быть дружественной классу `Student`, чтобы иметь доступ к члену `name`. (Но можно использовать открытый метод `Name()`, а не закрытый член `name`.)

Аналогично в функции можно было бы применять `valarray`-реализацию операции << для вывода — правда, ее нет. Поэтому для решения данной задачи в классе определен закрытый вспомогательный метод:

```
// Закрытый метод
ostream & Student::arr_out(ostream & os) const
{
   int i;
   int lim = scores.size();
   if (lim > 0)
   {
      for (i = 0; i < lim; i++)
      {
         os << scores[i] << " ";
         if (i % 5 == 4)
            os << endl;
      }
      if (i % 5 != 0)
         os << endl;
   }
   else
      os << " empty array ";
   return os;
}
```

Использование такой вспомогательной функции собирает в одном месте разбросанные фрагменты кода и делает код дружественной функции более аккуратным:

```
// использование версии операции operator<<() из класса string
ostream & operator<<(ostream & os, const Student & stu)
{
   os << "Scores for " << stu.name << ":\n";
   stu.arr_out(os);  // использование закрытого метода для scores
   return os;
}
```

При желании вспомогательную функцию можно применить и в качестве строительного блока для функций вывода пользовательского уровня.

В листинге 14.2 показан код методов класса `Student` — в том числе и методов, которые позволяют обращаться к отдельным оценкам из объекта `Student` с помощью операции `[]`.

Листинг 14.2. studentc.cpp

```
// studentc.cpp — класс Student, использующий включение
#include "studentc.h"
using std::ostream;
using std::endl;
using std::istream;
using std::string;

// Открытые методы
double Student::Average() const
{
   if (scores.size() > 0)
      return scores.sum()/scores.size();
   else
      return 0;
}
const string & Student::Name() const
{
   return name;
}
double & Student::operator[](int i)
{
   return scores[i];   // использует valarray<double>::operator[]()
}
double Student::operator[](int i) const
{
   return scores[i];
}

// Закрытый метод
ostream & Student::arr_out(ostream & os) const
{
   int i;
   int lim = scores.size();
   if (lim > 0)
   {
      for (i = 0; i < lim; i++)
      {
         os << scores[i] << " ";
         if (i % 5 == 4)
            os << endl;
      }
```

```
        if (i % 5 != 0)
            os << endl;
    }
    else
       os << " empty array ";
    return os;
}
// Друзья
// Использует версию operator>>() из класса string
istream & operator>>(istream & is, Student & stu)
{
    is >> stu.name;
    return is;
}
// Использует версию getline(ostream &, const string &) из класса string
istream & getline(istream & is, Student & stu)
{
    getline(is, stu.name);
    return is;
}
// Использует версию operator<<() из класса string
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << stu.name << ":\n";
    stu.arr_out(os);  // использование закрытого метода для scores
    return os;
}
```

За исключением закрытого вспомогательного метода, листинг 14.2 практически не требует написания нового кода. Включение позволяет использовать код, написанный ранее вами или кем-то еще.

Использование нового класса `Student`

Давайте напишем небольшую программу для тестирования класса `Student`. Для простоты она должна использовать массив из трех объектов `Student`, каждый из которых содержит пять экзаменационных оценок. Она должна использовать цикл ввода без усложнений вроде проверки вводимых значений, но не позволять преждевременно закончить ввод. Тестовая программа показана в листинге 14.3. Компилировать ее нужно вместе с `studentc.cpp`.

Листинг 14.3. `use_stuc.cpp`

```
// use_stuc.cpp -- использование составного класса
// Компилировать вместе с studentc.cpp
#include <iostream>
#include "studentc.h"
using std::cin;
using std::cout;
using std::endl;

void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;
int main()
{
    Student ada[pupils] =
        {Student(quizzes), Student(quizzes), Student(quizzes)};
```

```
    int i;
    for (i = 0; i < pupils; ++i)
        set(ada[i], quizzes);
    cout << "\nStudent List:\n";                          // вывод списка студентов
    for (i = 0; i < pupils; ++i)
        cout << ada[i].Name() << endl;
    cout << "\nResults:";                                 // вывод оценок
    for (i = 0; i < pupils; ++i)
    {
        cout << endl << ada[i];
        cout << "average: " << ada[i].Average() << endl;  // средняя оценка
    }
    cout << "Done.\n";
    return 0;
}
void set(Student & sa, int n)
{
    cout << "Please enter the student's name: ";         // ввод имени студента
    getline(cin, sa);
    cout << "Please enter " << n << " quiz scores:\n";   // ввод оценок для студента
    for (int i = 0; i < n; i++)
        cin >> sa[i];
    while (cin.get() != '\n')
        continue;
}
```

Ниже показан пример запуска программы, представленной в листингах 14.1, 14.2 и 14.3:

```
Please enter the student's name: Gil Bayts
Please enter 5 quiz scores:
92 94 96 93 95
Please enter the student's name: Pat Roone
Please enter 5 quiz scores:
83 89 72 78 95
Please enter the student's name: Fleur O'Day
Please enter 5 quiz scores:
92 89 96 74 64
Student List:
Gil Bayts
Pat Roone
Fleur O'Day

Results:
Scores for Gil Bayts:
92 94 96 93 95
average: 94

Scores for Pat Roone:
83 89 72 78 95
average: 83.4

Scores for Fleur O'Day:
92 89 96 74 64
average: 83
Done.
```

Закрытое наследование

В C++ имеется и другое средство реализации отношений *содержит* — *закрытое наследование*. При использовании закрытого наследования открытые (public) и защищенные (protected) члены базового класса становятся закрытыми членами производного класса — т.е. методы базового класса не переходят в открытый интерфейс производного объекта. Однако они могут использоваться внутри функций-членов производного класса.

Рассмотрим тему интерфейсов подробнее. При открытом наследовании открытые методы базового класса становятся открытыми методами производного класса. То есть производный класс наследует интерфейс базового класса. Это соответствует отношению *является*. А при закрытом наследовании открытые методы базового класса становятся закрытыми методами производного класса. То есть производный класс не наследует интерфейс базового класса. Как вы уже знаете, отсутствие наследования для включаемых объектов означает отношение *содержит*.

При закрытом наследовании класс наследует реализацию. Например, если базовым классом для класса Student является класс string, класс Student будет содержать компонент унаследованного класса string, который можно использовать для хранения строк. А методы класса Student могут использовать методы класса string для внутреннего доступа к компоненту string.

Включение добавляет в класс именованный объект-член, а закрытое наследование добавляет в класс неименованный унаследованный объект. В этой книге для обозначения объектов, добавленных путем наследования или включения, используется термин *подобъект*.

Значит, закрытое наследование обеспечивает те же свойства, что и включение — реализацию, но не интерфейс. Поэтому его также можно использовать для реализации отношения *содержит*. В действительности можно создать класс Student, использующий закрытое наследование и имеющий тот же открытый интерфейс, что и у версии с включением. То есть различия между двумя подходами влияют на реализацию, а не на интерфейс. Посмотрим, как можно переделать класс Student, используя закрытое наследование.

Новый вариант класса Student

Для получения закрытого интерфейса при определении класса необходимо использовать ключевое слово private вместо public. (На самом деле private принимается по умолчанию, поэтому отсутствие квалификатора доступа также приведет к закрытому наследованию.) Класс Student должен наследовать два класса, поэтому в объявлении класса Student следует указать их оба:

```
class Student : private std::string, private std::valarray<double>
{
public:
    ...
};
```

Наследование от нескольких базовых классов называется *множественным наследованием*. В общем случае множественное наследование — и особенно открытое множественное наследование — может привести к проблемам, которые устраняются с помощью дополнительных синтаксических правил. Мы поговорим об этом позже, но в данном случае проблем не будет.

В новом классе не нужны собственные закрытые данные, поскольку оба наследуемых базовых класса содержат все необходимые данные-члены. Версия этого примера с включением содержит в качестве членов два явно именованных объекта, а версия с закрытым наследованием содержит в качестве унаследованных членов два неименованных подобъекта. Это первое из главных отличий между двумя рассматриваемыми подходами.

Инициализация компонентов базового класса

Наличие неявно унаследованных компонентов вместо объектов-членов влияет на кодирование этого примера, поскольку для описания объекта уже нельзя использовать идентификаторы `name` и `score`. Вместо этого придется вернуться к технологии, применяемой при открытом наследовании. Конструктор, используемый при включении, имеет вид:

```
Student(const char * str, const double * pd, int n)
  : name(str), scores(pd, n) {}  // используются имена объектов для включения
```

В новой версии примера для наследуемых классов должен использоваться список инициализаторов членов, в котором для указания конструктора вместо имени *члена* применяется имя *класса*:

```
Student(const char * str, const double * pd, int n)
   : std::string(str), ArrayDb(pd, n) {}    // используются имена классов
                                            // для наследования
```

Здесь, как и в предыдущем примере, `ArrayDb` — это `typedef` для `std::valarray<double>`. Не забывайте, что список инициализаторов членов вместо `name(str)` использует определение `std::string(str)`. Это второе главное отличие между двумя рассматриваемыми подходами.

В листинге 14.4 приведено новое определение класса. Единственным отличием является отсутствие явно указанных имен объектов и применение имен классов вместо имен членов во встроенных конструкторах.

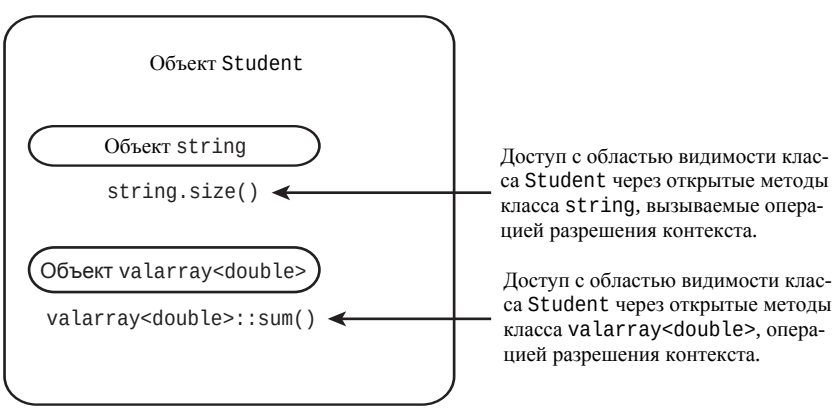
Листинг 14.4. `studenti.h`

```
// studenti.h -- определение класса Student через закрытое наследование
#ifndef STUDENTC_H_
#define STUDENTC_H_
#include <iostream>
#include <valarray>
#include <string>
class Student : private std::string, private std::valarray<double>
{
private:
   typedef std::valarray<double> ArrayDb;
   // Закрытый метод для вывода оценок
   std::ostream & arr_out(std::ostream & os) const;
public:
   Student() : std::string("Null Student"), ArrayDb() {}
   explicit Student(const std::string & s)
      : std::string(s), ArrayDb() {}
   explicit Student(int n) : std::string("Nully"), ArrayDb(n) {}
   Student(const std::string & s, int n)
      : std::string(s), ArrayDb(n) {}
   Student(const std::string & s, const ArrayDb & a)
      : std::string(s), ArrayDb(a) {}
```

```
    Student(const char * str, const double * pd, int n)
        : std::string(str), ArrayDb(pd, n) {}
    ~Student() {}

    double Average() const;
    double & operator[](int i);
    double operator[](int i) const;
    const std::string & Name() const;

    // Друзья
    // Ввод
    friend std::istream & operator>>(std::istream & is,
        Student & stu);         // 1 слово

    friend std::istream & getline(std::istream & is,
        Student & stu);         // 1 строка

    // Вывод
    friend std::ostream & operator<<(std::ostream & os,
        const Student & stu);
};
#endif
```

Доступ к методам базового класса

Закрытое наследование позволяет использовать методы базового класса только внутри методов производного класса. Но иногда необходимо обращаться к методам базового класса извне. Например, объявление класса `Student` предполагает возможность вызова функции `Average()`. При использовании включения для этого нужно вызывать методы `size()` и `sum()` класса `valarray` внутри открытой функции `Student::average()` (рис. 14.2).




```
class Student:private string,
private valarray<double>
{
    …
};
```

***Рис. 14.2.** Объекты внутри объектов: закрытое наследование*

При включении методы вызывались для объектов:

```
double Student::Average() const
{
   if (scores.size() > 0)
      return scores.sum()/scores.size();
   else
      return 0;
}
```

Однако здесь наследование позволяет применять имя класса и операцию разрешения контекста для вызова методов базовых классов:

```
double Student::Average() const
{
   if (ArrayDb::size() > 0)
      return ArrayDb::sum()/ArrayDb::size();
   else
      return 0;
}
```

В общем, при включении для вызова методов применяются имена объектов, а в случае закрытого наследования — имя класса и операция разрешения контекста.

Доступ к объектам базового класса

Операция разрешения контекста позволяет обращаться к методам базового класса. А что если нужен доступ к самому объекту базового класса? Например, в версии класса `Student` с включением метод `Name()` возвращает член `name` объекта `string`. Но при закрытом наследовании у объекта `string` нет имени. Как же код класса `Student` может обратиться к внутреннему объекту `string`?

Решением служит приведение типов. Тип `Student` порожден от `string`, поэтому объект `Student` можно привести к типу `string`. Вспомните, что указатель `this` указывает на вызвавший объект. Тогда `*this` является самим вызвавшим объектом, в данном случае — объектом `Student`. Чтобы не вызывать конструкторы для создания новых объектов, необходимо использовать приведение типа для создания ссылок:

```
const string & Student::Name() const
{
   return (const string &) *this;
}
```

Этот код возвращает ссылку на унаследованный объект `string`, который находится в вызывающем объекте `Student`.

Доступ к друзьям базового класса

Явное указание имени функции с именем ее класса не работает для дружественных функций, т.к. дружественная функция не принадлежит этому классу. Но для корректного вызова функций можно использовать явное приведение типа к базовому классу. В принципе, это та же техника, что и при доступе к объектам базового класса в методах класса. Но в случае с друзьями доступно имя объекта `Student`, поэтому вместо `*this` в коде используется имя объекта. Например, рассмотрим следующее определение дружественной функции:

```
ostream & operator<<(ostream & os, const Student & stu)
{
   os << "Scores for " << (const string &) stu << ":\n";
   ...
}
```

Если plato — объект типа Student, то показанный ниже оператор вызывает эту функцию, где stu — ссылка на plato, а os — ссылка на cout:

```
cout << plato;
```

Рассмотрим следующую строку кода:

```
os << "Scores for " << (const string &) stu << ":\n";
```

Приведение типа явно преобразует stu в ссылку на объект типа string, а этот тип вызывает функцию operator<<(ostream &, const string &). Ссылка stu не преобразуется автоматически в ссылку на string, поскольку при закрытом наследовании ссылке или указателю на базовый класс нельзя присвоить ссылку или указатель на производный класс без явного приведения типа.

Однако даже при открытом наследовании нужно иметь явные приведения типов. Одна из причин состоит в том, что без приведения код вроде показанного ниже соответствует прототипу дружественной функции, что приводит к рекурсивному вызову:

```
os << stu;
```

Существует еще одна причина: поскольку класс использует множественное наследование, компилятор не может определить, в какой базовый класс выполнять преобразование, т.к. оба базовых класса поддерживают функцию operator<<().

В листинге 14.5 показаны все методы класса Student, за исключением приведенных непосредственно в объявлении класса.

Листинг 14.5. studenti.cpp

```
// studenti.cpp -- класс Student, использующий закрытое наследование
#include "studenti.h"
using std::ostream;
using std::endl;
using std::istream;
using std::string;

// Открытые методы
double Student::Average() const
{
   if (ArrayDb::size() > 0)
      return ArrayDb::sum()/ArrayDb::size();
   else
      return 0;
}

const string & Student::Name() const
{
   return (const string &) *this;
}

double & Student::operator[](int i)
{
   return ArrayDb::operator[](i); // использование ArrayDb::operator[]()
}

double Student::operator[](int i) const
{
   return ArrayDb::operator[](i);
}
```

```
// Закрытый метод
ostream & Student::arr_out(ostream & os) const
{
    int i;
    int lim = ArrayDb::size();
    if (lim > 0)
    {
        for (i = 0; i < lim; i++)
        {
            os << ArrayDb::operator[](i) << " ";
            if (i % 5 == 4)
                os << endl;
        }
        if (i % 5 != 0)
            os << endl;
    }
    else
        os << " empty array ";
    return os;
}

// Друзья
// Использует версию operator>>() из класса string
istream & operator>>(istream & is, Student & stu)
{
    is >> (string &)stu;
    return is;
}

// Использует друга string - getline(ostream &, const string &)
istream & getline(istream & is, Student & stu)
{
    getline(is, (string &)stu);
    return is;
}

// Использует версию operator<<() из класса string
ostream & operator<<(ostream & os, const Student & stu)
{
    os << "Scores for " << (const string &) stu << ":\n";
    stu.arr_out(os); // использование закрытого метода для scores
    return os;
}
```

В этом примере также задействован код `string` и `valarray`, и поэтому дополнительное кодирование сведено к минимуму — за исключением закрытого вспомогательного метода.

Использование пересмотренного класса `Student`

Настало время протестировать новый класс. Обратите внимание, что обе версии класса `Student` имеют совершенно одинаковые открытые интерфейсы, поэтому работу обеих версий можно проверить с помощью одной и той же программы. Единственное отличие состоит в том, что нужно включить файл `studenti.h` вместо `studentc.h` и компоновать программу с файлом `studenti.cpp` вместо `studentc.cpp`. Код этой программы приведен в листинге 14.6. Не забудьте скомпилировать ее вместе с файлом `studenti.cpp`.

Листинг 14.6. use_stui.cpp

```
// use_stui.cpp -- использование класса с закрытым наследованием
// Компилировать вместе с studenti.cpp
#include <iostream>
#include "studenti.h"
using std::cin;
using std::cout;
using std::endl;
void set(Student & sa, int n);
const int pupils = 3;
const int quizzes = 5;

int main()
{
   Student ada[pupils] =
      {Student(quizzes), Student(quizzes), Student(quizzes)};
   int i;
   for (i = 0; i < pupils; i++)
      set(ada[i], quizzes);
   cout << "\nStudent List:\n";                          // вывод списка студентов
   for (i = 0; i < pupils; ++i)
      cout << ada[i].Name() << endl;
   cout << "\nResults:";                                 // вывод оценок
   for (i = 0; i < pupils; i++)
   {
      cout << endl << ada[i];
      cout << "average: " << ada[i].Average() << endl; // средняя оценка
   }
   cout << "Done.\n";
   return 0;
}
void set(Student & sa, int n)
{
   cout << "Please enter the student's name: ";        // ввод имени студента
   getline(cin, sa);
   cout << "Please enter " << n << " quiz scores:\n";  // ввод оценок для студента
   for (int i = 0; i < n; i++)
      cin >> sa[i];
   while (cin.get() != '\n')
      continue;
}
```

Ниже показан пример запуска программы из листинга 14.6:

```
Please enter the student's name: Gil Bayts
Please enter 5 quiz scores:
92 94 96 93 95
Please enter the student's name: Pat Roone
Please enter 5 quiz scores:
83 89 72 78 95
Please enter the student's name: Fleur O'Day
Please enter 5 quiz scores:
92 89 96 74 64

Student List:
Gil Bayts
Pat Roone
Fleur O'Day
```

```
Results:
Scores for Gil Bayts:
92 94 96 93 95
average: 94

Scores for Pat Roone:
83 89 72 78 95
average: 83.4

Scores for Fleur O'Day:
92 89 96 74 64
average: 83
Done.
```

При тех же входных данных, что и раньше, выходные данные программы совпадают с результатами версии с включением.

Включение или закрытое наследование?

Итак, отношение *содержит* можно смоделировать с помощью как включения, так и закрытого наследования — но что из них выбрать? Большинство программистов на C++ предпочитают включение. Во-первых, его проще проследить. В определении класса четко видны явно именованные объекты, которые представляют содержащиеся классы, и к этим объектам можно обращаться по именам. При наследовании же отношение выглядит более абстрактно. Во-вторых, наследование может приводить к трудностям, особенно, если класс наследуется от нескольких базовых классов. Может случиться так, что разные базовые классы содержат методы с одинаковыми именами, или у разных базовых классов общий предок. В общем, при использовании включения меньше вероятность столкнуться с проблемами. Включение позволяет иметь несколько подобъектов с одинаковыми именами. Если в классе нужны три объекта `string`, то с помощью включения можно определить три отдельных члена `string`. А наследование позволяет иметь только один объект — трудно различить объекты, у которых нет имени.

Однако у закрытого наследования есть возможности, недостижимые при включении. Предположим, что класс содержит защищенные члены, которые могут быть данными или функциями. Такие члены доступны для производных классов, но не для всего мира. Если включить этот класс в другой класс с помощью композиции, новый класс будет как раз частью всего мира, а не наследником, и поэтому не будет иметь доступ к защищенным членам. Но если использовать наследование, то новый класс будет наследником, а значит, сможет обращаться к защищенным членам.

Еще одна ситуация, в которой удобно закрытое наследование — переопределение виртуальных функций. Это привилегия производных, а не содержащих классов. При закрытом наследовании переопределенные функции могут применяться только внутри класса.

Совет

В общем случае для создания отношения *содержит* нужно использовать включение. Если новому классу нужен доступ к защищенным членам базовых классов или требуется переопределить виртуальную функцию, необходимо закрытое наследование.

Защищенное наследование

Защищенное наследование – это разновидность закрытого наследования. В этом случае при объявлении базового класса указывается ключевое слово `protected`:

```
class Student : protected std::string,
                protected std::valarray<double>
{...};
```

При защищенном наследовании открытые и защищенные члены базового класса становятся защищенными членами производного класса. Как и при закрытом наследовании, интерфейс базового класса доступен в производном классе, но не доступен внешнему миру. Главное отличие между защищенным и закрытым наследованием проявляется при порождении от класса, который сам является производным классом. При закрытом наследовании класс третьего поколения не получает доступа к интерфейсу базовых классов. Это происходит потому, что открытые методы базового класса становятся закрытыми в производном классе, и доступ к закрытым членам и методам из следующего уровня наследования невозможен. При защищенном наследовании открытые методы базового класса становятся защищенными членами производного класса и доступны внутри классов следующего уровня наследования.

В табл. 14.1 приведены свойства открытого, закрытого и защищенного наследования. Выражение *неявное восходящее приведение* означает, что указатель или ссылка на базовый класс может ссылаться на объект производного класса без явного приведения типа.

Таблица 14.1. Разновидности наследования

Свойство	Открытое наследование	Защищенное наследование	Закрытое наследование
Открытые члены становятся	Открытыми членами производного класса	Защищенными членами производного класса	Закрытыми членами производного класса
Защищенные члены становятся	Защищенными членами производного класса	Защищенными членами производного класса	Закрытыми членами производного класса
Закрытые члены становятся	Доступными только через интерфейс базового класса	Доступными только через интерфейс базового класса	Доступными только через интерфейс базового класса
Неявное восходящее преобразование	Да	Да, но только внутри производного класса	Нет

Переопределение доступа с помощью `using`

При защищенном или закрытом порождении открытые члены базового класса становятся защищенными или закрытыми. Предположим, что нужно создать какой-то метод базового класса, открытый в производном классе. Одна из возможностей – определить метод производного класса, использующий метод базового класса. Пусть, например, нужно, чтобы класс `Student` мог использовать метод `sum()` из `valarray`. Можно объявить в определении класса метод `sum()`, а затем определить его следующим образом:

```
double Student::sum() const                    // открытый метод Student
{
   return std::valarray<double>::sum();        // использует закрытый
                                               // унаследованный метод
}
```

Тогда объект Student может вызывать метод Student::sum(), который, в свою очередь, применяет метод valarray<double>::sum() к встроенному объекту valarray. (Если в области видимости находится typedef по имени ArrayDb, то вместо std::valarray<double> можно использовать описатель ArrayDb.)

Вместо упаковки одной функции в другую можно использовать другой способ — объявление using (наподобие тех, которые применяются в пространствах имен), согласно которому конкретный метод базового класса может использоваться производным классом, даже если наследование является закрытым.

Предположим, что необходимо применять методы min() и max() из класса valarray с классом Student. В этом случае в раздел public файла studenti.h можно добавить объявление using:

```
class Student : private std::string, private std::valarray<double>
{
   ...
public:
   using std::valarray<double>::min;
   using std::valarray<double>::max;
   ...
};
```

Объявление using делает методы valarray<double>::min() и valarray<double>::max() доступными, как будто это открытые методы класса Student:

```
cout << "high score: " << ada[i].max() << endl;
```

Обратите внимание, что объявление using использует только имя члена — без скобок, сигнатуры функции и возвращаемого типа. Например, чтобы сделать метод operator[]() из valarray доступным в классе Student, нужно поместить в раздел private определения класса Student такое объявление using:

```
using std::valarray<double>::operator[];
```

После этого будут доступны обе версии — с квалификатором const и без него. Затем можно удалить существующие прототипы и определения для Student::operator[](). Объявление using работает только в случае наследования и не работает при технике включения.

Имеется и более старый способ переопределения методов базового класса в классе, порожденном с помощью закрытого наследования — нужно поместить имя метода в разделе public класса-наследника, например:

```
class Student : private std::string, private std::valarray<double>
{
public:
   std::valarray<double>::operator[];  // переопределен как открытый,
                                       // достаточно указать имя
   ...
};
```

Это похоже на объявление using, но без ключевого слова using. Применять такой подход *не рекомендуется*, поскольку он считается устаревшим. Поэтому если ваш компилятор поддерживает объявление using, лучше использовать его, чтобы делать методы базовых классов доступными для классов-наследников.

Множественное наследование

Множественное наследование описывает класс, у которого есть несколько базовых классов. Подобно одиночному наследованию, открытое множественное наследование выражает отношение *является*. Например, если имеются классы `Waiter` и `Singer`, то от них можно породить класс `SingingWaiter`:

```
class SingingWaiter : public Waiter, public Singer {...};
```

Учтите, что оба базовых класса должны сопровождаться ключевым словом `public`, т.к. по умолчанию компилятор подразумевает закрытое наследование:

```
class SingingWaiter : public Waiter, Singer {...}; // Singer считается закрытым
                                                   // базовым классом
```

Как было сказано ранее в этой главе, закрытое и защищенное множественное наследование могут выражать отношение *содержит*; примером служит реализация класса `Student` в файле `studenti.h`. А мы сейчас займемся открытым наследованием.

Множественное наследование может привнести новые проблемы при программировании. Две главные из них — наследование разных методов с одинаковыми именами от разных базовых классов и наследование нескольких экземпляров класса от нескольких взаимосвязанных базовых классов. Для устранения этих проблем введены новые правила и варианты синтаксиса. Таким образом, использование множественного наследования может оказаться более сложным и предрасположенным к проблемам, чем одиночное наследование. По этой причине многие члены сообщества C++ весьма настороженно относятся к множественному наследованию, а некоторые хотели бы вообще изъять его из языка. Другим множественное наследование, наоборот, нравится, и они считают его удобным и даже необходимым для отдельных проектов. Третьи советуют применять множественное наследование аккуратно и умеренно.

Рассмотрим конкретный пример вместе с сопутствующими проблемами и способами их устранения. Для множественного наследования необходимо несколько классов. В этом примере мы определим абстрактный базовый класс `Worker` (Работник) и породим от него классы `Waiter` (Официант) и `Singer` (Певец). Потом, используя множественное наследование, от классов `Waiter` и `Singer` мы породим класс `SingingWaiter` (Поющий официант; рис. 14.3). В этом случае базовый класс `Worker` наследуется через два различных его потомка, что вызывает наибольшие трудности при использовании множественного наследования. Начнем с определений классов `Worker`, `Waiter` и `Singer`, которые представлены в листинге 14.7.

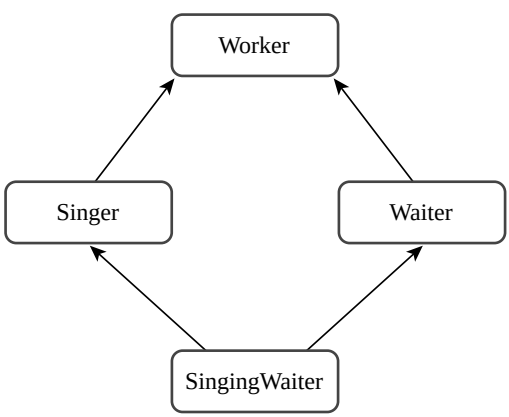


***Рис. 14.3.** Множественное наследование с общим предком*

Листинг 14.7. worker0.h

```
// worker0.h -- классы работников
#ifndef WORKER0_H_
#define WORKER0_H_
#include <string>
class Worker                    // работник – абстрактный базовый класс
{
private:
   std::string fullname;
   long id;
public:
   Worker() : fullname("no one"), id(0L) {}
   Worker(const std::string & s, long n)
            : fullname(s), id(n) {}
   virtual ~Worker() = 0;       // чистый виртуальный деструктор
   virtual void Set();
   virtual void Show() const;
};
class Waiter : public Worker    // официант
{
private:
   int panache;
public:
   Waiter() : Worker(), panache(0) {}
   Waiter(const std::string & s, long n, int p = 0)
            : Worker(s, n), panache(p) {}
   Waiter(const Worker & wk, int p = 0)
            : Worker(wk), panache(p) {}
   void Set();
   void Show() const;
};
class Singer : public Worker    // певец
{
protected:
   enum {other, alto, contralto, soprano, bass, baritone, tenor};
   enum {Vtypes = 7};
private:
   static char *pv[Vtypes];     // строковые эквиваленты видов голоса
   int voice;
public:
   Singer() : Worker(), voice(other) {}
   Singer(const std::string & s, long n, int v = other)
            : Worker(s, n), voice(v) {}
   Singer(const Worker & wk, int v = other)
            : Worker(wk), voice(v) {}
   void Set();
   void Show() const;
};
#endif
```

Объявления классов в листинге 14.7 включают ряд внутренних констант, представляющих виды голоса. Перечисление определяет символические константы `alto`, `contralto` и т.д., а в статическом массиве `pv` хранятся указатели на строковые эквиваленты в стиле C. В файле реализации, приведенном в листинге 14.8, осуществляется инициализация этого массива и содержатся определения методов.

Листинг 14.8. worker0.cpp

```
// worker0.cpp -- методы классов работников
#include "worker0.h"
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// Методы Worker
// Виртуальный деструктор должен быть реализован, даже если он является чистым
Worker::~Worker() {}

void Worker::Set()
{
   cout << "Enter worker's name: ";  // ввод имени и фамилии работчика
   getline(cin, fullname);
   cout << "Enter worker's ID: ";    // ввод идентификатора работчика
   cin >> id;
   while (cin.get() != '\n')
      continue;
}

void Worker::Show() const
{
   cout << "Name: " << fullname << "\n";           // имя и фамилия
   cout << "Employee ID: " << id << "\n";          // идентификатор
}

// Методы Waiter
void Waiter::Set()
{
   Worker::Set();
   cout << "Enter waiter's panache rating: ";
         // Ввод индекса элегантности официанта
   cin >> panache;
   while (cin.get() != '\n')
      continue;
}

void Waiter::Show() const
{
   cout << "Category: waiter\n";                  // категория: официант
   Worker::Show();
   cout << "Panache rating: " << panache << "\n";  // индекс элегантности
}

// Методы Singer
char * Singer::pv[] = {"other", "alto", "contralto",
                  "soprano", "bass", "baritone", "tenor"};
void Singer::Set()
{
   Worker::Set();
   cout << "Enter number for singer's vocal range:\n";
         // Ввод номера вокального диапазона певца
   int i;
   for (i = 0; i < Vtypes; i++)
   {
      cout << i << ": " << pv[i] << " ";
      if ( i % 4 == 3)
         cout << endl;
   }
   if (i % 4 != 0)
      cout << endl;
```

```
    while (cin >> voice && (voice < 0 || voice >= Vtypes) )
        cout << "Please enter a value >= 0 and < " << Vtypes << endl;
    while (cin.get() != '\n')
        continue;
}
void Singer::Show() const
{
    cout << "Category: singer\n";                    // категория: певец
    Worker::Show();
    cout << "Vocal range: " << pv[voice] << endl;    // вокальный диапазон
}
```

Код в листинге 14.9 предназначен для краткого тестирования классов с использованием полиморфного массива указателей.

Листинг 14.9. `worktest.cpp`

```
// worktest.cpp -- тестирование иерархии классов сотрудников
#include <iostream>
#include "worker0.h"
const int LIM = 4;
int main()
{
    Waiter bob("Bob Apple", 314L, 5);
    Singer bev("Beverly Hills", 522L, 3);
    Waiter w_temp;
    Singer s_temp;
    Worker * pw[LIM] = {&bob, &bev, &w_temp, &s_temp};
    int i;
    for (i = 2; i < LIM; i++)
        pw[i]->Set();
    for (i = 0; i < LIM; i++)
    {
        pw[i]->Show();
        std::cout << std::endl;
    }
    return 0;
}
```

Ниже показан вывод программы из листингов 14.7, 14.8 и 14.9:

```
Enter waiter's name: Waldo Dropmaster
Enter worker's ID: 442
Enter waiter's panache rating: 3
Enter singer's name: Sylvie Sirenne
Enter worker's ID: 555
Enter number for singer's vocal range:
0: other 1: alto 2: contralto 3: soprano
4: bass 5: baritone 6: tenor
3
Category: waiter
Name: Bob Apple
Employee ID: 314
Panache rating: 5

Category: singer
Name: Beverly Hills
Employee ID: 522
Vocal range: soprano
```

```
Category: waiter
Name: Waldo Dropmaster
Employee ID: 442
Panache rating: 3

Category: singer
Name: Sylvie Sirenne
Employee ID: 555
Vocal range: soprano
```

Вроде бы все работает: указатели на `Waiter` вызывают `Waiter::Show()` и `Waiter::Set()`, а указатели на `Singer` – `Singer::Show()` и `Singer::Set()`. Тем не менее, трудности возникают, если нужно добавить класс `SingingWaiter`, порожденный от двух классов `Singer` и `Waiter`. В частности, возникают два следующих вопроса.

- Сколько всего сотрудников?
- Какой использовать метод?

Сколько всего сотрудников?

Начнем с открытого наследования `SingingWaiter` от классов `Singer` и `Waiter`:

```
class SingingWaiter: public Singer, public Waiter {...};
```

Поскольку и `Singer`, и `Waiter` наследуют компонент `Worker`, то `SingingWaiter` будет иметь два компонента `Worker` (рис. 14.4).

```
class Singer : public Worker { ...};
class Waiter : public Worker { ...};
class SingingWaiter : public Singer, public Waiter { ...};
```

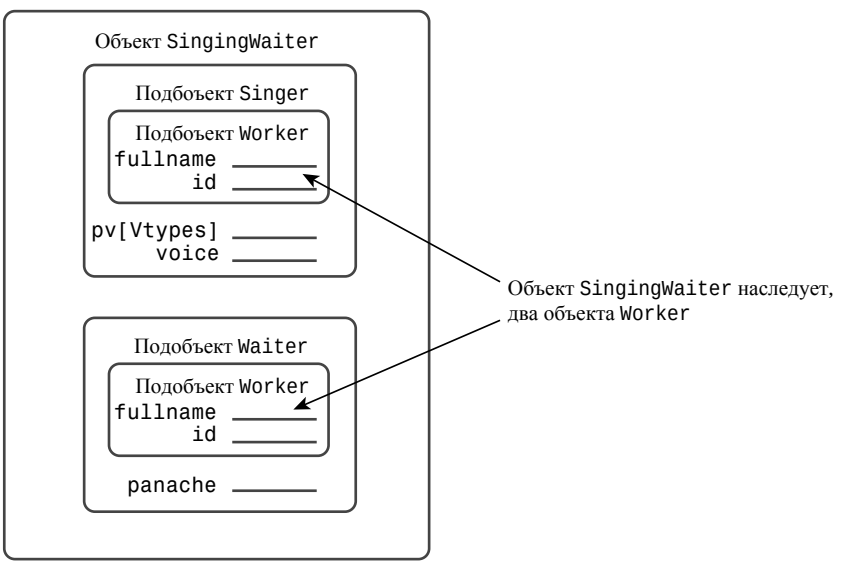


***Рис. 14.4.** Наследование двух объектов базового класса*

Понятно, что это приводит к трудностям. Например, в обычной ситуации указателю на базовый класс можно присвоить адрес объекта производного класса, но теперь эта операция неоднозначна:

```
SingingWaiter ed;
Worker * pw = &ed;                    // неоднозначность
```

Обычно такое присваивание заносит в указатель на базовый класс адрес объекта базового класса внутри производного объекта. Но ed содержит два объекта Worker, из которых нужно выбрать один. Конкретный объект можно указать с помощью приведения типа:

```
Worker * pw1 = (Waiter *) &ed;        // Worker из Waiter
Worker * pw2 = (Singer *) &ed;        // Worker из Singer
```

Это, безусловно, усложняет использование массива указателей на базовый класс для ссылок на множество объектов (полиморфизм).

Наличие двух копий объектов Worker приводит и к другим сложностям. Но основной вопрос таков: зачем вообще нужны две копии объекта Worker? Поющая официантка, как и любой другой работник, должна иметь только одно имя и один идентификатор. С введением множественного наследования в C++ появились виртуальные базовые классы, делающие такое наследование возможным.

Виртуальные базовые классы

Виртуальные базовые классы позволяют объекту, порожденному от нескольких базовых классов, которые сами имеют общий базовый класс, наследовать только один объект от этого базового класса. Например, можно сделать класс Worker виртуальным базовым классом для Singer и Waiter, указав в определениях класса ключевое слово virtual (virtual и public можно использовать в любом порядке):

```
class Singer : virtual public Worker {...};
class Waiter : public virtual Worker {...};
```

Затем необходимо определить SingingWaiter, как и раньше:

```
class SingingWaiter: public Singer, public Waiter {...};
```

Теперь объект SingingWaiter будет содержать лишь одну копию объекта Worker, а производные объекты Singer и Waiter будут иметь один общий базовый объект Worker вместо двух его копий (рис. 14.5). Поскольку объект SingingWaiter теперь содержит один подобъект Worker, можно снова использовать полиморфизм.

Рассмотрим несколько возможных вопросов.

- Почему используется термин *виртуальный*?
- Почему нельзя обойтись без объявления базовых классов виртуальными и сделать виртуальное поведение нормой при множественном наследовании?
- Есть ли здесь какие-то опасности?

Итак, первое: почему используется термин "виртуальный"? Связь между концепциями виртуальных функций и виртуальных базовых классов совсем не очевидна. Оказывается, сообщество пользователей C++ очень не любит вводить новые ключевые слова. Будет неудобно, например, если новое ключевое слово совпадет с именем важной функции или переменной в популярной программе. Поэтому в C++ ключевое слово virtual просто используется для новой возможности — своего рода "перегрузка" ключевого слова.

Далее, почему нельзя обойтись без определения базовых классов виртуальными и сделать виртуальное поведение нормой для множественного наследования? Во-первых, бывают случаи, когда нужно иметь несколько копий базового класса.

```
class Singer : virtual public Worker { ...};
class Waiter : virtual public Worker { ...};
class SingingWaiter : public Singer, public Waiter { ...};
```

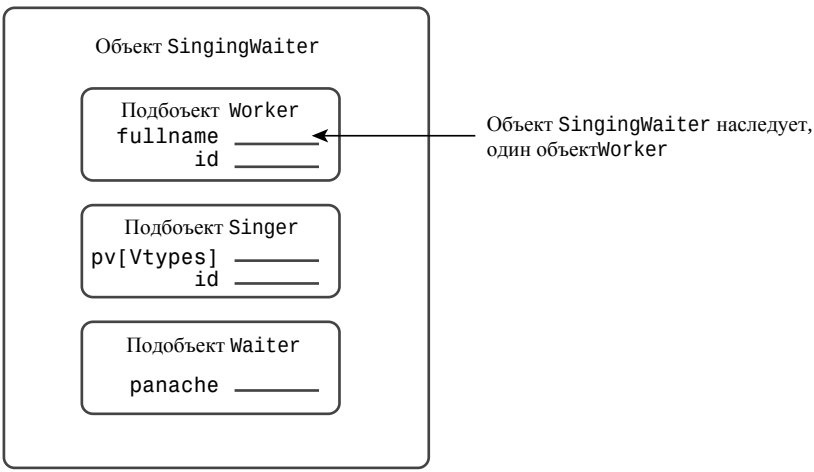


Рис. 14.5. *Наследование с виртуальными базовыми классами*

Во-вторых, если сделать базовые классы виртуальными, программа будет выполнять дополнительную работу — а зачем платить за то, что вам не нужно? В-третьих, существуют трудности, которые будут описаны в следующем абзаце.

Наконец, есть ли опасности? Да, есть. Использование виртуальных базовых классов требует изменения правил C++, и некоторые вещи придется кодировать по-другому. Кроме того, применение виртуальных базовых классов потребует изменения существующего кода. Например, при добавлении класса `SingingWaiter` в иерархию класса `Worker` придется вернуться назад и добавить ключевое слово `virtual` в определения классов `Singer` и `Waiter`.

Новые правила для конструкторов

Наличие виртуальных базовых классов требует нового подхода к конструкторам класса. При использовании невиртуальных базовых классов в списке инициализации могут присутствовать *только* конструкторы непосредственных базовых классов. Однако эти конструкторы, в свою очередь, могут передавать информацию своим базовым классам. Например, возможна следующая организация конструкторов:

```
class A
{
   int a;
public:
   A(int n = 0) : a(n) {}
   ...
};
class B: public A
{
   int b;
public:
   B(int m = 0, int n = 0): A(n), b(m) {}
   ...
};
```

```
class C : public B
{
   int c;
public:
   C(int q = 0, int m = 0, int n = 0) : B(m, n), c(q) {}
   ...
};
```

Конструктор класса C может вызывать только конструкторы класса B, а конструктор B может вызывать только конструкторы из класса A. В примере конструктор C использует значение q и передает значения m и n обратно конструктору B. Конструктор B использует значение m и передает значение n конструктору A.

Но если Worker будет виртуальным базовым классом, автоматическая передача информации работать не будет. Рассмотрим, к примеру, следующий конструктор для случая множественного наследования:

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
              : Waiter(wk,p), Singer(wk,v) {}  // неверно
```

Проблема в том, что в данном случае wk автоматически передается в объект Worker двумя разными путями (через Waiter и Singer). Для устранения возможного конфликта C++ отключает автоматическую передачу информации через промежуточный класс в базовый класс, *если* он виртуальный. Поэтому вышеприведенный конструктор инициализирует члены panache и voice, однако аргумент wk *не будет* передан в подобъект Waiter. Однако компилятор должен создать компонент базового объекта перед созданием производных компонентов — в данном случае он будет использовать конструктор по умолчанию Worker.

Если для создания виртуального базового класса нужен не конструктор по умолчанию, придется явно вызывать соответствующий базовый конструктор. Поэтому конструктор должен выглядеть так:

```
SingingWaiter(const Worker & wk, int p = 0, int v = Singer::other)
              : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
```

Здесь явно вызывается конструктор Worker(const Worker &). Такое использование допустимо, а зачастую и необходимо, для виртуальных базовых классов, но ошибочно для невиртуальных базовых классов.

Внимание!

Если у класса есть непрямой виртуальный базовый класс, конструктор этого класса должен явно вызывать конструктор виртуального базового класса, за исключением случаев, когда достаточно конструктора по умолчанию виртуального базового класса.

Какой метод использовать?

Помимо изменений в правилах для конструкторов классов, множественное наследование часто требует и других перенастроек в исходном коде. Рассмотрим задачу расширения метода Show() для класса SingingWaiter. Поскольку у объекта SingingWaiter нет новых членов данных, можно подумать, что достаточно использовать унаследованные методы. Однако это порождает первую проблему. Предположим, что новой версии Show() нет, и попытаемся использовать объект SingingWaiter для вызова унаследованного метода Show():

```
SingingWaiter newhire("Elise Hawks", 2005, 6, soprano);
newhire.Show();  // неоднозначность
```

При одиночном наследовании отсутствие переопределения функции Show() приводит к использованию самого последнего наследственного определения этой функции. Но в данном случае у каждого прямого предка имеется метод Show(), что делает этот вызов неоднозначным.

Внимание!

Множественное наследование может приводить к неоднозначным вызовам функций. Например, класс BadDude может наследовать два совершенно разных метода Draw() от классов Gunslinger и PokerPlayer.

Для ясности можно применить операцию разрешения контекста:

```
SingingWaiter newhire("Elise Hawks", 2005, 6, soprano);
newhire.Singer::Show();  // использование версии Singer
```

Но лучше переопределить в классе SingingWaiter метод Show(), указав, какую версию Show() следует использовать. Например, если нужно, чтобы SingingWaiter пользовался версией из Singer, можно сделать так:

```
void SingingWaiter::Show()
{
   Singer::Show();
}
```

Такой способ вызова базового метода из производного метода нормально работает для одиночного наследования. Предположим, например, что от класса Waiter (Официант) порожден класс HeadWaiter (Старший официант). Можно использовать следующую последовательность определений, где каждый производный класс добавляет к информации базового класса вывод своей дополнительной информации:

```
void Worker::Show() const
{
   cout << "Name: " << fullname << "\n";
   cout << "Employee ID: " << id << "\n";
}
void Waiter::Show() const
{
   Worker::Show();
   cout << "Panache rating: " << panache << "\n";
}
void HeadWaiter::Show() const
{
   Waiter::Show();
   cout << "Presence rating: " << presence << "\n";
}
```

Но для SingingWaiter такой способ не сработает. Показанный метод даст сбой, поскольку он игнорирует компонент Waiter:

```
void SingingWaiter::Show()
{
   Singer::Show();
}
```

Это можно исправить, вызвав еще и версию из Waiter:

```
void SingingWaiter::Show()
{
   Singer::Show();
   Waiter::Show();
}
```

Но тогда имя и идентификатор сотрудника будут выведены дважды, т.к. и `Singer::Show()`, и `Waiter::Show()` вызывают `Worker::Show()`.

Как исправить положение? Один из способов — воспользоваться модульным подходом вместо инкрементного. То есть нужно определить метод, выводящий только компоненты `Worker`, затем метод, выводящий только компоненты `Waiter` (вместо компонентов `Waiter` плюс `Worker`), и, наконец, метод, выводящий компоненты `Singer`. Потом необходимо собрать эти компоненты вместе в методе `SingingWaiter::Show()`. Например, можно сделать так:

```
void Worker::Data() const
{
   cout << "Name: " << fullname << "\n";
   cout << "Employee ID: " << id << "\n";
}
void Waiter::Data() const
{
   cout << "Panache rating: " << panache << "\n";
}
void Singer::Data() const
{
   cout << "Vocal range: " << pv[voice] << "\n";
}
void SingingWaiter::Data() const
{
   Singer::Data();
   Waiter::Data();
}
void SingingWaiter::Show() const
{
   cout << "Category: singing waiter\n";
   Worker::Data();
   Data();
}
```

Аналогично можно построить и другие методы `Show()` из соответствующих компонентов `Data()`.

При таком подходе объекты по-прежнему вызывают метод `Show()` открытым образом. Но методы `Data()` должны быть внутренними в классах — т.е. вспомогательными методами, обеспечивающими открытый интерфейс. Однако если сделать методы `Data()` закрытыми, то вызов `Worker::Data()` из кода `Waiter` будет невозможным. Это как раз одна из ситуаций, где могут пригодиться защищенные классы. Если методы `Data()` являются защищенными, их можно использовать внутри всех классов иерархии, но извне они будут недоступны.

Другой способ — сделать не закрытыми, а защищенными все компоненты данных. Однако использование защищенных методов вместо защищенных данных позволяет более точно управлять доступом к данным.

Методы `Set()`, запрашивающие данные для установки значений объекта, представляют похожую проблему. Например, вызов `SingingWaiter::Set()` должен запрашивать информацию для объекта `Worker` один раз, а не два. Такое же решение необходимо и для `Show()`. Можно определить защищенные методы `Get()`, которые запрашивают информацию только для одного класса, а затем объединить методы `Set()`, использующие методы `Get()` в качестве строительных блоков.

Итак, введение множественного наследования с общим предком требует построения виртуальных базовых классов, изменения правил для списков инициализации в конструкторах и, возможно, переделки классов, если они написаны с учетом множественного наследования. В листинге 14.10 приведены измененные определения классов, а в листинге 14.11 — их реализации.

Листинг 14.10. workermi.h

```
// workermi.h -- классы сотрудников с множественным наследованием
#ifndef WORKERMI_H_
#define WORKERMI_H_
#include <string>
class Worker                          // абстрактный базовый класс
{
private:
   std::string fullname;
   long id;
protected:
   virtual void Data() const;
   virtual void Get();
public:
   Worker() : fullname("no one"), id(0L) {}
   Worker(const std::string & s, long n)
          : fullname(s), id(n) {}
   virtual ~Worker() = 0;             // чистая виртуальная функция
   virtual void Set() = 0;
   virtual void Show() const = 0;
};
class Waiter : virtual public Worker
{
private:
   int panache;
protected:
   void Data() const;
   void Get();
public:
   Waiter() : Worker(), panache(0) {}
   Waiter(const std::string & s, long n, int p = 0)
          : Worker(s, n), panache(p) {}
   Waiter(const Worker & wk, int p = 0)
          : Worker(wk), panache(p) {}
   void Set();
   void Show() const;
};
class Singer : virtual public Worker
{
protected:
   enum {other, alto, contralto, soprano,
        bass, baritone, tenor};
   enum {Vtypes = 7};
   void Data() const;
   void Get();
private:
   static char *pv[Vtypes];           // строковые эквиваленты видов голосов
   int voice;
public:
   Singer() : Worker(), voice(other) {}
   Singer(const std::string & s, long n, int v = other)
          : Worker(s, n), voice(v) {}
```

```
    Singer(const Worker & wk, int v = other)
           : Worker(wk), voice(v) {}
    void Set();
    void Show() const;
};
// Множественное наследование
class SingingWaiter : public Singer, public Waiter
{
protected:
    void Data() const;
    void Get();
public:
    SingingWaiter() {}
    SingingWaiter(const std::string & s, long n, int p = 0,
                int v = other)
           : Worker(s,n), Waiter(s, n, p), Singer(s, n, v) {}
    SingingWaiter(const Worker & wk, int p = 0, int v = other)
           : Worker(wk), Waiter(wk,p), Singer(wk,v) {}
    SingingWaiter(const Waiter & wt, int v = other)
           : Worker(wt),Waiter(wt), Singer(wt,v) {}
    SingingWaiter(const Singer & wt, int p = 0)
           : Worker(wt),Waiter(wt,p), Singer(wt) {}
    void Set();
    void Show() const;
};
#endif
```

Листинг 14.11. workermi.cpp

```
// workermi.cpp -- методы классов работников с множественным наследованием
#include "workermi.h"
#include <iostream>
using std::cout;
using std::cin;
using std::endl;
// Методы Worker
Worker::~Worker() { }
// Защищенные методы
void Worker::Data() const
{
    cout << "Name: " << fullname << endl;        // имя и фамилия
    cout << "Employee ID: " << id << endl;       // идентификатор
}
void Worker::Get()
{
    getline(cin, fullname);
    cout << "Enter worker's ID: ";               // ввод идентификатора работчика
    cin >> id;
    while (cin.get() != '\n')
        continue;
}
// Методы Waiter
void Waiter::Set()
{
    cout << "Enter waiter's name: ";             // ввод имени и фамилии работчика
    Worker::Get();
    Get();
}
```

```
void Waiter::Show() const
{
   cout << "Category: waiter\n";                  // категория: официант
   Worker::Data();
   Data();
}
// Защищенные методы
void Waiter::Data() const
{
   cout << "Panache rating: " << panache << endl;  // индекс элегантности
}
void Waiter::Get()
{
   cout << "Enter waiter's panache rating: ";
        // Ввод индекса элегантности официанта
   cin >> panache;
   while (cin.get() != '\n')
      continue;
}
// Методы Singer
char * Singer::pv[Singer::Vtypes] = {"other", "alto", "contralto",
                                     "soprano", "bass", "baritone", "tenor"};
void Singer::Set()
{
   cout << "Enter singer's name: ";               // Ввод имени и фамилии певца
   Worker::Get();
   Get();
}
void Singer::Show() const
{
   cout << "Category: singer\n";                  // Категория: певец
   Worker::Data();
   Data();
}
// Защищенные методы
void Singer::Data() const
{
   cout << "Vocal range: " << pv[voice] << endl;   // Вокальный диапазон
}
void Singer::Get()
{
   cout << "Enter number for singer's vocal range:\n";
        // Ввод номера вокального диапазона певца
   int i;
   for (i = 0; i < Vtypes; i++)
   {
      cout << i << ": " << pv[i] << " ";
      if ( i % 4 == 3)
         cout << endl;
   }
   if (i % 4 != 0)
      cout << '\n';
   cin >> voice;
   while (cin.get() != '\n')
      continue;
}
```

```
// Методы SingingWaiter
void SingingWaiter::Data() const
{
   Singer::Data();
   Waiter::Data();
}
void SingingWaiter::Get()
{
   Waiter::Get();
   Singer::Get();
}
void SingingWaiter::Set()
{
   cout << "Enter singing waiter's name: ";
        // Ввод имени и фамилии поющего официанта
   Worker::Get();
   Get();
}
void SingingWaiter::Show() const
{
   cout << "Category: singing waiter\n";  // категория: поющий официант
   Worker::Data();
   Data();
}
```

Конечно, нужно протестировать эти классы — хотя бы из любопытства. В листинге 14.12 приведен необходимый для этого код. Обратите внимание, что в программе при присваивании адресов различных видов классов указателям на базовые классы используется полиморфизм. Применяется также функция `strchr()` из библиотеки работы со строкам в стиле C:

```
while (strchr("wstq", choice) == NULL)
```

Эта функция возвращает адрес первого вхождения символа `choice` в строку `"wstq"` (если символ не найден, возвращается указатель `NULL`). Такая проверка проще, чем оператор `if` для сравнения с `choice` каждого символа.

Не забудьте скомпилировать листинг 14.12 с файлом `workermi.cpp`.

Листинг 14.12. `workmi.cpp`

```
// workmi.cpp -- множественное наследование
// компилировать вместе с workermi.cpp
#include <iostream>
#include <cstring>
#include "workermi.h"
const int SIZE = 5;
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   using std::strchr;
   Worker * lolas[SIZE];
   int ct;
   for (ct = 0; ct < SIZE; ct++)
   {
      char choice;
      cout << "Enter the employee category:\n"    // ввод категории работника;
```

```
            << "w: waiter s: singer "          // w — официант, s — певец,
            << "t: singing waiter q: quit\n"; // t - поющий официант, q - завершение
        cin >> choice;
        while (strchr("wstq", choice) == NULL)
        {
            cout << "Please enter a w, s, t, or q: ";
            cin >> choice;
        }
        if (choice == 'q')
            break;
        switch(choice)
        {
            case 'w': lolas[ct] = new Waiter;
                      break;
            case 's': lolas[ct] = new Singer;
                      break;
            case 't': lolas[ct] = new SingingWaiter;
                      break;
        }
        cin.get();
        lolas[ct]->Set();
    }
    cout << "\nHere is your staff:\n";          // вывод списка работников
    int i;
    for (i = 0; i < ct; i++)
    {
        cout << endl;
        lolas[i]->Show();
    }
    for (i = 0; i < ct; i++)
        delete lolas[i];
    cout << "Bye.\n";
    return 0;
}
```

Ниже показан пример выполнения программы из листингов 14.10, 14.11 и 14.12

```
Enter the employee category:
w: waiter s: singer t: singing waiter q: quit
w
Enter waiter's name: Wally Slipshod
Enter worker's ID: 1040
Enter waiter's panache rating: 4
Enter the employee category:
w: waiter s: singer t: singing waiter q: quit
s
Enter singer's name: Sinclair Parma
Enter worker's ID: 1044
Enter number for singer's vocal range:
0: other 1: alto 2: contralto 3: soprano
4: bass 5: baritone 6: tenor
5
Enter the employee category:
w: waiter s: singer t: singing waiter q: quit
t
Enter singing waiter's name: Natasha Gargalova
Enter worker's ID: 1021
Enter waiter's panache rating: 6
```

```
Enter number for singer's vocal range:
0: other 1: alto 2: contralto 3: soprano
4: bass 5: baritone 6: tenor
3
Enter the employee category:
w: waiter s: singer t: singing waiter q: quit
q

Here is your staff:

Category: waiter
Name: Wally Slipshod
Employee ID: 1040
Panache rating: 4

Category: singer
Name: Sinclair Parma
Employee ID: 1044
Vocal range: baritone

Category: singing waiter
Name: Natasha Gargalova
Employee ID: 1021
Vocal range: soprano
Panache rating: 6
Bye.
```

Давайте рассмотрим еще насколько вопросов, касающихся множественного наследования.

Смесь виртуальных и невиртуальных базовых классов

Снова рассмотрим случай с производным классом, который наследует базовый класс несколькими путями. Если базовый класс виртуальный, то производный класс содержит один подобъект базового класса. Если базовый класс невиртуальный, производный класс содержит несколько подобъектов. А если базовые классы смешанные? Предположим, что класс `B` является виртуальным базовым классом для классов `C` и `D`, и невиртуальным базовым классом для классов `X` и `Y`. Предположим также, что класс `M` порожден от классов `C`, `D`, `X` и `Y`. В этом случае класс `M` содержит один подобъект класса `B` для всех виртуальных унаследованных предков (классов `C` и `D`) и по отдельному подобъекту класса `B` для каждого невиртуального предка (классов `X` и `Y`). Значит, всего он будет содержать три подобъекта класса `B`. При наследовании одного базового класса несколькими виртуальными и несколькими невиртуальными путями производный класс будет содержать один объект базового класса для представления всех виртуальных путей и отдельные объекты базового класса для каждого невиртуального пути.

Виртуальные базовые классы и доминирование

Использование виртуальных базовых классов меняет механизм разрешения неоднозначностей в C++. Для невиртуальных базовых классов правила просты. Если класс наследует несколько членов (данных или методов) с одинаковыми именами от различных классов, указание такого имени без квалификатора класса приведет к неоднозначности. Однако если задействованы виртуальные базовые классы, то использование имени без квалификатора класса может и не быть неоднозначным. Если одно имя *доминирует* над остальными, его можно применять без квалификатора класса.

Каким образом имя одного члена может доминировать над другим? Имя в производном классе доминирует над такими же именами из классов-предков, независимо от того, родители это или более дальние предки. Рассмотрим следующие определения:

```
class B
{
public:
   short q();
   ...
};
class C : virtual public B
{
public:
   long q();
   int omg();
   ...
};
class D : public C
{
   ...
};
class E : virtual public B
{
private:
   int omg();
   ...
};
class F: public D, public E
{
   ...
};
```

Здесь определение `q()` из класса `C` доминирует над таким же определением из класса `B`, поскольку `C` порожден от `B`. Значит, методы класса `F` могут использовать запись `q()` для вызова `C::q()`. С другой стороны, ни одно определение `omg()` не может доминировать над другими, поскольку ни `C`, ни `E` не являются базовыми классами друг для друга. Поэтому попытка использовать в классе `F` вызов `omg()` без квалификатора класса приведет к неоднозначности.

Правила виртуальной неоднозначности не считаются с правилами доступа. То есть хотя `E::omg()` является закрытым и поэтому недоступным непосредственно для класса `F`, запись `omg()` будет неоднозначной. Аналогично, если даже `C::q()` будет закрытым, он будет доминировать над `D::q()`. В этом случае в классе `F` возможен вызов `B::q()`, но неуточненный `q()` будет ссылаться на недоступный метод `C::q()`.

Краткий обзор множественного наследования

Сначала вспомним множественное наследование без виртуальных базовых классов. Эта форма множественного наследования не вводит новых правил. Однако если класс наследует два члена с одинаковыми именами, но от разных классов, то в производном классе для различения этих двух членов необходимо использовать квалификаторы класса. Так, методы класса `BadDude`, унаследованные от `Gunslinger` и `PokerPlayer`, должны применять `Gunslinger::draw()` и `PokerPlayer::draw()` для различения методов `draw()`, унаследованных от двух разных классов. В противном случае компилятор выдаст сообщение о неоднозначном обращении.

Если производный класс наследуется от невиртуального базового класса несколькими путями, то он наследует по одному объекту базового класса для каждого экземпляра базового класса. В некоторых случаях так и нужно, но чаще наличие нескольких экземпляров базового класса приводит к проблемам.

Теперь рассмотрим множественное наследование с виртуальными базовыми классами. Класс становится виртуальным базовым классом, когда в производном классе при описании наследования используется ключевое слово `virtual`:

```
class marketing : public virtual reality { ... };
```

Главное отличие и причина применения виртуальных базовых классов заключается в том, что класс, порождаемый от одного или более экземпляров виртуального базового класса, наследует только один объект базового класса. Реализация этот свойства приводит к следующим требованиям.

- В производном классе с непрямым виртуальным базовым классом конструкторы должны непосредственно вызывать конструкторы непрямых базовых классов, что недопустимо для непрямых виртуальных базовых классов.
- Неоднозначность имен разрешается в соответствии с правилами доминирования.

Как видите, множественное наследование может приводить к сложностям в программировании. Правда, эти сложности возникают тогда, когда производный класс наследуется несколькими путями от одного и того же базового класса. Если не ввязываться в такие ситуации, то остается лишь при необходимости уточнять унаследованные имена.

Шаблоны классов

Наследование (открытое, закрытое и защищенное) и включение не всегда являются решением, позволяющим повторно использовать код. Рассмотрим, например, класс `Stack` (см. главу 10) и класс `Queue` (см. главу 12). Это примеры *контейнерных классов*, содержащих другие типы объектов или данных. Класс `Stack` из главы 10, например, содержит значения `unsigned long`. Легко можно создать стек для хранения значений `double` или объектов `sting`. Код будет аналогичным, за исключением типов хранимых объектов. Но вместо того чтобы определять новые классы, хорошо было бы определить стек в общей (не зависящей от типа) форме и задавать конкретные типы как параметры класса. Тогда один и тот же общий код можно будет использовать для создания стеков разных типов величин. В главе 10 в примере `Stack` в качестве первого шага в достижении этой цели используется конструкция `typedef`. Но такому подходу присуща пара недостатков. Во-первых, при каждом изменении типа придется редактировать заголовочный файл. Во-вторых, этот прием позволяет создать лишь один вид стека на программу. Ведь `typedef` не может определять два разных типа одновременно, и поэтому невозможно так создать одновременно, скажем, стек значений `int` и стек объектов `string`.

Шаблоны классов в C++ предлагают более подходящий способ создания обобщенных определений класса. (Изначально язык C++ не поддерживал шаблоны, а с момента появления они постоянно развиваются. Поэтому если используется устаревший компилятор, могут поддерживаться не все рассматриваемые здесь возможности.) Шаблоны предоставляют *параметризованные* типы — т.е. при создании класса или функции можно передать имя типа в качестве аргумента. Передав, к примеру, в шаблон `Queue` имя типа `int`, можно указать компилятору создать класс `Queue` для хранения в очереди целых значений.

Библиотека C++ содержит несколько шаблонов классов. Ранее в этой главе рассматривался шаблон класса `valarray`, а в главе 4 были описаны шаблонные классы `vector` и `array`. Стандартная библиотека шаблонов (Standard Template Library – STL) C++, частично рассматриваемая в главе 16, предлагает мощные и гибкие реализации шаблонов для нескольких контейнерных классов. В этой главе мы ознакомимся с построением более простых конструкций.

Определение шаблона класса

В качестве модели для построения шаблона воспользуемся классом `Stack` из главы 10. Вот исходное определение класса:

```
typedef unsigned long Item;

class Stack
{
private:
    enum {MAX = 10};                // константа, характерная для класса
    Item items[MAX];                // содержит элементы стека
    int top;                        // индекс вершины стека
public:
    Stack();
    bool isempty() const;
    bool isfull() const;

    // push() возвращает false, если стек полон, и true - в противном случае
    bool push(const Item & item);   // добавляет элемент в стек

    // pop() возвращает false, если стек пуст, и true - в противном случае
    bool pop(Item & item);          // выталкивает элемент с вершины стека
};
```

При построении шаблона определение класса `Stack` заменяется определением шаблона, а функции-члены класса – функциями-членами шаблона. Как и для шаблонных функций, шаблонный класс предваряется следующим кодом:

```
template <class Тип>
```

Ключевое слово `template` сообщает компилятору, что далее следует определение шаблона. Часть кода в угловых скобках аналогична списку аргументов в функции. Можно считать, что ключевое слово `class` служит именем типа для переменной, которая получает тип как значение, а *Тип* является именем этой переменной.

Слово `class` не означает, что *Тип* должно быть классом; это означает только, что *Тип* служит в качестве спецификатора обобщенного типа, который будет заменен реальным типом при использовании шаблона. Последние реализации C++ позволяют применять вместо `class` более точное ключевое слово `typename`:

```
template <typename Тип>             // новый вариант
```

Вместо *Тип* можно вписать свое имя обобщенного типа; правила именования здесь такие же, что и для любого другого идентификатора. Обычно используют идентификаторы `T` и `Type`; мы будем применять `Type`. При вызове шаблона параметр *Тип* заменяется конкретным типом вроде `int` или `string`. Внутри определения шаблона имя обобщенного типа можно использовать для указания типа, который будет храниться в стеке. В случае класса `Stack` нужно указывать `Type` везде, где в старом определении применялся идентификатор `Item` из конструкции `typedef`. Например:

```
Item items[MAX];                    // содержит элементы стека
```

станет выглядеть как

```
Type items[MAX];              // содержит элементы стека
```

Аналогично можно заменить методы исходного класса функциями-членами шаблона. Каждая функция должна предваряться таким же объявлением шаблона:

```
template <class Type>
```

Здесь также необходимо заменить идентификатор `Item` из конструкции `typedef` именем обобщенного типа `Type`. Кроме того, квалификатор класса `Stack::` нужно заменить вариантом `Stack<Type>::`. Например:

```
bool Stack::push(const Item & item)
{
    ...
}
```

преобразуется в:

```
template <class Type>         // или template <typename Type>
bool Stack<Type>::push(const Type & item)
{
    ...
}
```

Если внутри определения класса определить метод (встроенное определение), можно опустить преамбулу шаблона и квалификатор класса.

В листинге 14.13 приведены комбинированные шаблоны класса и функций-членов. Важно понимать, что эти шаблоны не являются определениями классов и функций-членов. Это скорее указания компилятору C++, как сгенерировать определения класса и функций-членов. Конкретная актуализация шаблона — например, класс стека для управления объектами `string` — называется *созданием экземпляра* или *специализацией*. Шаблонные функции-члены нельзя размещать в отдельном файле реализации. (Одно время в стандарте языка существовало ключевое слово `export`, которое позволяло такой вынос в отдельный файл реализации. Однако оно не было учтено в очень многих реализациях. В C++11 это слово уже не входит в стандарт, но зарезервировано для возможного дальнейшего использования.) Поскольку шаблоны не являются функциями, их нельзя компилировать отдельно. Шаблоны необходимо применять совместно с запросами на создание экземпляров шаблонов. Проще всего это сделать, поместив всю информацию о шаблонах в заголовочный файл и включив этот заголовочный файл в файл, использующий шаблоны.

Листинг 14.13. `stacktp.h`

```
// stacktp.h -- шаблон стека
#ifndef STACKTP_H_
#define STACKTP_H_
template <class Type>

class Stack
{
private:
    enum {MAX = 10};          // константа, специфичная для класса
    Type items[MAX];          // содержит элементы стека
    int top;                  // индекс вершины стека
public:
    Stack();
    bool isempty();
```

```
   bool isfull();
   bool push(const Type & item);        // добавление item в стек
   bool pop(Type & item);               // выталкивание из стека в item
};

template <class Type>
Stack<Type>::Stack()
{
   top = 0;
}

template <class Type>
bool Stack<Type>::isempty()
{
   return top == 0;
}

template <class Type>
bool Stack<Type>::isfull()
{
   return top == MAX;
}

template <class Type>
bool Stack<Type>::push(const Type & item)
{
   if (top < MAX)
   {
      items[top++] = item;
      return true;
   }
   else
      return false;
}

template <class Type>
bool Stack<Type>::pop(Type & item)
{
   if (top > 0)
   {
      item = items[--top];
      return true;
   }
   else
      return false;
}

#endif
```

Использование шаблонного класса

Просто включение шаблона в программу не генерирует шаблонный класс — необходимо запросить создание экземпляра. Для этого потребуется объявить объект с типом шаблонного класса и заменить имя обобщенного типа конкретным типом. Например, ниже показано создание двух стеков: одного для хранения значений `int`, а другого — для объектов `string`.

```
   Stack<int> kernels;                  // создание стека для значений int
   Stack<string> colonels;              // создание стека для объектов string
```

Встретив эти два определения, компилятор на основе шаблона `Stack<Type>` сгенерирует два различных объявления класса и два разных набора методов класса. Объявление класса `Stack<int>` заменит все `Type` на `int`, а объявление класса `Stack<string>` заменит все `Type` на `string`. Конечно, используемые алгоритмы должны согласоваться с типами. К примеру, класс `Stack` предполагает, что возможно присваивание одних элементов другим. Это присваивание справедливо для базовых типов, структур и классов (если не сделать операцию присваивания закрытой), но не для массивов.

Идентификаторы обобщенных типов, такие как `Type` в нашем примере, называются *параметрами типа* — т.е. они действуют примерно как переменные, но вместо присваивания числового значения параметру типа присваивается тип. Поэтому в объявлении `kernels` параметр `Type` имеет значение `int`.

Учтите, что требуемый тип необходимо указать явно. В этом заключается отличие от обычных шаблонных функций — в них компилятор может использовать типы аргументов для определения вида функции, которую нужно сгенерировать:

```
template <class T>
void simple(T t) { cout << t << '\n'; }
...
simple(2);                // генерирует void simple(int)
simple("two");            // генерирует void simple(const char *)
```

В листинге 14.14 приведена первоначальная программа тестирования работы стека (см. листинг 10.12), но приспособленная для использования строкового представления идентификаторов заказов вместо значений `unsigned long`.

Листинг 14.14. `stacktem.cpp`

```
// stacktem.cpp -- тестирование шаблонного класса стека
#include <iostream>
#include <string>
#include <cctype>
#include "stacktp.h"
using std::cin;
using std::cout;
int main()
{
   Stack<std::string> st;                          // создание пустого стека
   char ch;
   std::string po;
   cout << "Please enter A to add a purchase order,\n"  // A - добавить заказ,
        << "P to process a PO, or Q to quit.\n";    // P - обработать заказ, Q - выйти
   while (cin >> ch && std::toupper(ch) != 'Q')
   {
      while (cin.get() != '\n')
         continue;
      if (!std::isalpha(ch))
      {
         cout << '\a';
         continue;
      }
      switch(ch)
      {
         case 'A':
         case 'a': cout << "Enter a PO number to add: ";
                        // Ввод номера добавляемого заказа
```

```
                cin >> po;
                if (st.isfull())
                    cout << "stack already full\n";         // стек уже полон
                else
                    st.push(po);
                break;
        case 'P':
        case 'p': if (st.isempty())
                    cout << "stack already empty\n";        // стек уже пуст
                else {
                    st.pop(po);
                    cout << "PO #" << po << " popped\n";    // заказ извлечен
                    break;
                }
    }
    cout << "Please enter A to add a purchase order,\n"
         << "P to process a PO, or Q to quit.\n";
  }
  cout << "Bye\n";
  return 0;
}
```

Ниже приведен пример запуска программы из листинга 14.14:

```
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: red911porsche
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: blueR8audi
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
A
Enter a PO number to add: silver747boeing
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #silver747boeing popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #blueR8audi popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
PO #red911porsche popped
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
P
stack already empty
Please enter A to add a purchase order,
P to process a PO, or Q to quit.
Q
Bye
```

Более внимательный взгляд на шаблонные классы

В качестве типа для шаблона класса `Stack<Type>` можно использовать встроенный тип или объект класса. А как насчет указателей? Например, можно ли применить в листинге 14.14 не объект `sting`, а указатель на `char`? В конце концов, такие указатели являются встроенным средством для работы со строками в стиле C. Ответ таков: конечно, можно создать стек указателей, но он будет плохо работать без существенной переделки программ. Компилятор может создать какой угодно класс, однако задача программиста — правильно его использовать. Сначала посмотрим, почему такой стек указателей будет плохо работать с кодом из листинга 14.14, а затем рассмотрим пример полезного применения стека указателей.

Некорректное использование стека указателей

Давайте кратко рассмотрим три простых, но неудачных попытки адаптации листинга 14.14 к использованию стека указателей. Из этих примеров необходимо извлечь урок, чтобы в дальнейшем при создании шаблонов не действовать вслепую. Все три примера начинаются с совершенно допустимого вызова шаблона `Stack<Type>`:

```
Stack<char *> st;  // создание стека указателей на символы
```

Вариант 1: оператор

```
string po;
```

из листинга 14.14 меняется на

```
char * po;
```

Смысл в том, чтобы для ввода данных с клавиатуры использовать указатель на `char` вместо объекта `string`. Этот подход ошибочен с самого начала — ведь одно только создание указателя не выделяет память для хранения входных строк. (Программа скомпилируется нормально, но, скорее всего, завершится аварийно, когда `cin` попытается сохранить введенные данные в неподходящем месте.)

Вариант 2: оператор

```
string po;
```

заменяется на

```
char po[40];
```

Здесь выделяется память для входной строки, а переменная `po` имеет тип `char *`, и поэтому ее можно поместить в стек. Однако массив ведет себя совершенно не так, как нужно в методе `pop()`:

```
template <class Type>
bool Stack<Type>::pop(Type & item)
{
   if (top > 0)
   {
      item = items[--top];
      return true;
   }
   else
      return false;
}
```

Во-первых, ссылочная переменная `item` должна ссылаться на некоторого вида lvalue, но не на имя массива. Во-вторых, предполагается, что переменной `item` можно

присваивать значения. Даже если бы переменная item могла ссылаться на массив, невозможно присвоить значение имени массива. Так что этот способ тоже не годится.

Вариант 3: оператор

```
string po;
```

заменяется на

```
char * po = new char[40];
```

Здесь выделяется память для входной строки, а po является переменной и поэтому совместима с кодом метода pop(). Однако здесь мы сталкиваемся с наиболее фундаментальной проблемой: имеется только одна переменная po, и она всегда указывает на одно и тоже место в памяти. Правда, содержимое памяти меняется при каждом чтении новой строки, но каждая операция заталкивания помещает в стек один и тот же адрес. Поэтому при выталкивании данных из стека мы всегда будем получать один и тот же адрес, и он всегда будет указывать на последнюю строку, прочитанную и сохраненную в памяти. Такой стек не сохраняет отдельно каждую новую строку по мере их ввода и поэтому бесполезен.

Корректное использование стека указателей

Один из способов применения стека указателей — создание в вызывающей программе массива указателей, где все указатели указывают на разные строки. Помещение таких указателей в стек имеет смысл, т.к. они ссылаются на разные строки. Обратите внимание, что создание различных указателей — обязанность вызывающей программы, а не стека. Стек должен просто манипулировать готовыми указателями, а не создавать их.

Предположим, что нужно смоделировать следующую ситуацию. Секретарь привез преподавателю тележку с объемными курсовыми работами студентов. Если входной ящик преподавателя пуст, он берет из тележки верхнюю работу и кладет во входной ящик. Если входной ящик заполнен, преподаватель берет из него верхнюю работу, проверяет ее и кладет в выходной ящик. Если входной ящик заполнен частично, преподаватель может проверить верхнюю работу из входного ящика, а может взять верхнюю работу из тележки и положить во входной ящик. Чтобы решить, как поступить в каждом таком случае, он просто подбрасывает монетку. Попытаемся исследовать влияние его действий на первоначальный порядок курсовых работ.

Описанную ситуацию можно смоделировать с помощью массива указателей на строки, представляющие курсовые работы в тележке. Каждая строка содержит имя студента, написавшего работу. Для представления входного ящика можно использовать стек, а для представления выходного ящика — еще один массив указателей. Добавление работы во входной ящик можно представить заталкиванием указателя из входного массива в стек, а обработку папки — выталкиванием элемента из стека и добавлением его в выходной ящик.

Учитывая важность исследования всех аспектов данной задачи, будет полезно иметь возможность опробовать разные размеры стека. В листинге 14.15 класс Stack<Type> слегка переопределен так, чтобы конструктор Stack принимал размер стека в качестве аргумента. Это приводит к внутреннему использованию динамического массива, поэтому классу теперь требуется деструктор, конструктор копирования и операция присваивания. Кроме того, определение сокращает объем кода, т.к. некоторые методы встроены в код определения.

Листинг 14.15. stcktp1.h

```
// stcktp1.h -- модифицированный шаблон Stack
#ifndef STCKTP1_H_
#define STCKTP1_H_

template <class Type>
class Stack
{
private:
   enum {SIZE = 10};                // размер по умолчанию
   int stacksize;
   Type * items;                    // хранит элементы стека
   int top;                         // индекс вершины стека
public:
   explicit Stack(int ss = SIZE);
   Stack(const Stack & st);
   ~Stack() { delete [] items; }
   bool isempty() { return top == 0; }
   bool isfull() { return top == stacksize; }
   bool push(const Type & item);    // добавление item в стек
   bool pop(Type & item);           // выталкивание верхнего элемента в item
   Stack & operator=(const Stack & st);
};

template <class Type>
Stack<Type>::Stack(int ss) : stacksize(ss), top(0)
{
   items = new Type [stacksize];
}

template <class Type>
Stack<Type>::Stack(const Stack & st)
{
   stacksize = st.stacksize;
   top = st.top;
   items = new Type [stacksize];
   for (int i = 0; i < top; i++)
      items[i] = st.items[i];
}

template <class Type>
bool Stack<Type>::push(const Type & item)
{
   if (top < stacksize)
   {
      items[top++] = item;
      return true;
   }
   else
      return false;
}

template <class Type>
bool Stack<Type>::pop(Type & item)
{
   if (top > 0)
   {
      item = items[--top];
      return true;
   }
```

```
    else
        return false;
}

template <class Type>
Stack<Type> & Stack<Type>::operator=(const Stack<Type> & st)
{
    if (this == &st)
        return *this;
    delete [] items;
    stacksize = st.stacksize;
    top = st.top;
    items = new Type [stacksize];
    for (int i = 0; i < top; i++)
        items[i] = st.items[i];
    return *this;
}

#endif
```

Обратите внимание, что прототип объявляет тип, возвращаемый функцией операции присваивания, как ссылку на `Stack`, а само определение шаблонной функции задает тип как `Stack<Type>`. Первое объявление является сокращением для второго, но может использоваться только внутри области видимости класса. То есть можно применять тип `Stack` внутри определения шаблонов и шаблонных функций, а за пределами класса — например, при указании возвращаемых типов и использовании операции разрешения контекста — необходима полная форма `Stack<Type>`.

Программа в листинге 14.16 использует новый шаблон стека для моделирования действий преподавателя. Как и в предыдущих примерах, для генерации случайных чисел в ней используются функции `rand()`, `srand()` и `time()`. Случайно сгенерированные 0 и 1 моделируют подбрасывание монеты.

Листинг 14.16. `stkoptr1.cpp`

```
// stkoptr1.cpp -- тестирование стека указателей
#include <iostream>
#include <cstdlib>                              // для rand(), srand()
#include <ctime>                                // для time()
#include "stcktp1.h"
const int Num = 10;

int main()
{
    std::srand(std::time(0));                   // рандомизация rand()
    std::cout << "Please enter stack size: ";   // ввод размера стека
    int stacksize;
    std::cin >> stacksize;

    // Создание пустого стека размером stacksize
    Stack<const char *> st(stacksize);

    // Входной ящик
    const char * in[Num] = {
           " 1: Hank Gilgamesh", " 2: Kiki Ishtar",
           " 3: Betty Rocker", " 4: Ian Flagranti",
           " 5: Wolfgang Kibble", " 6: Portia Koop",
           " 7: Joy Almondo", " 8: Xaverie Paprika",
           " 9: Juan Moore", "10: Misha Mache"
    };
```

```
    // Выходной ящик
    const char * out[Num];
    int processed = 0;
    int nextin = 0;
    while (processed < Num)
    {
        if (st.isempty())
            st.push(in[nextin++]);
        else if (st.isfull())
            st.pop(out[processed++]);
        else if (std::rand() % 2 && nextin < Num)    // шансы 50 на 50
            st.push(in[nextin++]);
        else
            st.pop(out[processed++]);
    }
    for (int i = 0; i < Num; i++)
        std::cout << out[i] << std::endl;
    std::cout << "Bye\n";
    return 0;
}
```

Ниже приведены два примера запуска программы из листинга 14.16 (из-за случайного выбора конечный порядок работ может существенно изменяться даже при одинаковом размере стека):

```
Please enter stack size: 5
2: Kiki Ishtar
1: Hank Gilgamesh
3: Betty Rocker
5: Wolfgang Kibble
4: Ian Flagranti
7: Joy Almondo
9: Juan Moore
8: Xaverie Paprika
6: Portia Koop
10: Misha Mache
Bye

Please enter stack size: 5
3: Betty Rocker
5: Wolfgang Kibble
6: Portia Koop
4: Ian Flagranti
8: Xaverie Paprika
9: Juan Moore
10: Misha Mache
7: Joy Almondo
2: Kiki Ishtar
1: Hank Gilgamesh
Bye
```

Замечания по программе

Строки в программе, представленной в листинге 14.16, никуда не перемещаются. При заталкивании строки в стек просто создается новый указатель на уже существующую строку. То есть создается указатель с адресом существующей строки. При выталкивании строки из стека этот адрес копируется в выходной массив.

В программе используется тип const char *, т.к. массив указателей инициализируется набором строковых констант.

Как воздействует деструктор стека на строки? Никак. Конструктор класса использует операцию new для создания массива, содержащего указатели. Деструктор класса уничтожает этот массив, а не строки, на которые ссылаются элементы массива.

Пример шаблона массива и нетипизированные аргументы

Шаблоны часто используются для контейнерных классов, поскольку идея параметров типа удачно сочетается с идеей общего способа хранения для различных типов. На самом деле стремление предоставить повторно используемый код для контейнерных классов и было главной причиной введения шаблонов. Рассмотрим другой пример и исследуем несколько новых аспектов разработки и применения шаблонов. А именно, рассмотрим нетипизированные аргументы, или аргументы-выражения, и применение массива для управления семейством наследования.

Начнем с простого шаблона массива, который позволяет задавать размер массива. Один из приемов, который был использован в последней версии шаблона Stack — использование динамического массива внутри класса и аргумента в конструкторе для задания количества элементов. Другой подход состоит в применении аргумента шаблона для задания размера обычного массива, и как раз так поступает новый шаблон array в C++11. В листинге 14.17 приведена более скромная версия.

Листинг 14.17. arraytp.h

```
// arraytp.h -- шаблон массива
#ifndef ARRAYTP_H_
#define ARRAYTP_H_
#include <iostream>
#include <cstdlib>
template <class T, int n>
class ArrayTP
{
private:
   T ar[n];
public:
   ArrayTP() {};
   explicit ArrayTP(const T & v);
   virtual T & operator[](int i);
   virtual T operator[](int i) const;
};
template <class T, int n>
ArrayTP<T,n>::ArrayTP(const T & v)
{
   for (int i = 0; i < n; i++)
      ar[i] = v;
}
template <class T, int n>
T & ArrayTP<T,n>::operator[](int i)
{
   if (i < 0 || i >= n)
   {
      std::cerr << "Error in array limits: " << i  // выход за пределы допустимого
                << " is out of range\n";           // диапазона индекса в массиве
      std::exit(EXIT_FAILURE);
   }
   return ar[i];
}
```

```
template <class T, int n>
T ArrayTP<T,n>::operator[](int i) const
{
   if (i < 0 || i >= n)
   {
      std::cerr << "Error in array limits: " << i  // выход за пределы допустимого
                << " is out of range\n";           // диапазона индекса в массиве
      std::exit(EXIT_FAILURE);
   }
   return ar[i];
}
#endif
```

Обратите внимание на заголовок шаблона в листинге 14.17:

```
template <class T, int n>
```

Ключевое слово `class` (или эквивалентное в этом контексте `typename`) объявляет `T` как параметр типа, или аргумент типа. Ключевое слово `int` объявляет, что `n` имеет тип `int`. Такой вид параметра — определяющий конкретный тип, а не обобщенное имя типа — называется *нетипизированным параметром*, или *параметром-выражением*. Предположим, что имеется следующее определение:

```
ArrayTP<double, 12> eggweights;
```

Встретив его, компилятор определит класс `ArrayTP<double,12>` и создаст объект `eggweights` этого класса. При определении класса компилятор заменит `T` на `double` и `n` на 12.

Аргументы-выражения имеют некоторые ограничения. Аргумент-выражение может быть целочисленного типа, перечислимого типа, ссылкой или указателем. Поэтому объявление `double m` является недопустимым, тогда как `double & rm` и `double * pm` допускаются. Кроме того, код шаблона не может изменять значение аргумента или использовать его адрес. Например, в шаблоне `ArrayTP` выражения `n++` или `&n` не разрешены. При инициализации шаблона значение, используемое для аргумента-выражения, должно быть константным выражением.

Такой способ установки размера массива обладает одним преимуществом перед вариантом с конструктором, применяемым в `Stack`. Вариант с конструктором использует память типа кучи, управляемую операциями `new` и `delete`, а вариант с аргументом-выражением — стек памяти для автоматических переменных. Второй способ быстрее, особенно в случае множества небольших массивов.

Главный недостаток подхода с аргументами-выражениями состоит в том, что для каждого размера массива генерируется собственный шаблон. Так, следующие объявления генерируют два отдельных определения классов:

```
ArrayTP<double, 12> eggweights;
ArrayTP<double, 13> donuts;
```

Однако объявления, показанные ниже, генерируют только одно определение класса, а информация о размере передается конструктору этого класса:

```
Stack<int> eggs(12);
Stack<int> dunkers(13);
```

Другое отличие заключается в том, что вариант с конструктором более гибок, поскольку размер массива хранится как член класса, а не жестко закодирован в определении. Поэтому можно, например, определить присваивание массива одного размера массиву другого размера или создать класс с массивами переменной размерности.

Универсальность шаблонов

В шаблонных классах можно применять те же приемы программирования, что и в обычных классах. Шаблонные классы могут выступать как в качестве базовых классов, так и компонентов других классов.

Например, можно создать шаблон стека на основе шаблона массива. Или можно взять шаблон массива и применить его для создания массива, элементы которого являются стеками, основанными на шаблоне стека. Это значит, что возможен такой код:

```
template <typename T>          // или <class T>
class Array
{
private:
   T entry;
   ...
};

template <typename Type>
class GrowArray : public Array<Type> {...}; // наследование

template <typename Tp>
class Stack
{
   Array<Tp> ar;               // использует Array<> в качестве компонента
   ...
};

...
Array < Stack<int> > asi;      // массив стеков значений int
```

В последнем операторе во избежание путаницы с операцией >>, в C++98 требуется разделять два символа > как минимум одним пробельным символом. В C++11 это требование отсутствует.

Рекурсивное использование шаблонов

Другим примером универсальности шаблонов является возможность рекурсивного использования шаблонов. Например, приведенное ранее определение шаблона массива можно использовать так:

```
ArrayTP<ArrayTP<int,5>, 10> twodee;
```

Здесь создается массив `twodee`, состоящий из 10 элементов, каждый из которых, в свою очередь, является массивом из пяти целых чисел (`int`). Эквивалентный обычный массив объявляется следующим образом:

```
int twodee[10][5];
```

Обратите внимание, что в синтаксисе шаблона размеры массива приведены в порядке, отличном от эквивалентного обычного двумерного массива. Эта идея проверяется в листинге 14.18. Также в нем для создания одномерного массива, содержащего суммы и средние значения для каждого из десяти наборов по пять чисел, применяется шаблон `ArrayTP`.

Вызов метода `cout.width(2)` приводит к выводу следующего элемента массива в виде двух символов (если для вывода целого числа не потребуется бо́льшая длина).

Листинг 14.18. twod.cpp

```
// twod.cpp -- создание двумерного массива
#include <iostream>
#include "arraytp.h"

int main(void)
{
   using std::cout;
   using std::endl;
   ArrayTP<int, 10> sums;
   ArrayTP<double, 10> aves;
   ArrayTP<ArrayTP<int,5>, 10> twodee;
   int i, j;
   for (i = 0; i < 10; i++)
   {
      sums[i] = 0;
      for (j = 0; j < 5; j++)
      {
         twodee[i][j] = (i + 1) * (j + 1);
         sums[i] += twodee[i][j];
      }
      aves[i] = (double) sums[i] / 10;
   }
   for (i = 0; i < 10; i++)
   {
      for (j = 0; j < 5; j++)
      {
         cout.width(2);
         cout << twodee[i][j] << ' ';
      }
      cout << ": sum = ";
      cout.width(3);
      cout << sums[i] << ", average = " << aves[i] << endl;
   }
   cout << "Done.\n";
   return 0;
}
```

Вывод программы из листинга 14.18 содержит по одной строке для каждого из 10 элементов twodee, которые представляют собой массивы из пяти элементов:

```
 1  2  3  4  5 : sum =  15, average = 1.5
 2  4  6  8 10 : sum =  30, average = 3
 3  6  9 12 15 : sum =  45, average = 4.5
 4  8 12 16 20 : sum =  60, average = 6
 5 10 15 20 25 : sum =  75, average = 7.5
 6 12 18 24 30 : sum =  90, average = 9
 7 14 21 28 35 : sum = 105, average = 10.5
 8 16 24 32 40 : sum = 120, average = 12
 9 18 27 36 45 : sum = 135, average = 13.5
10 20 30 40 50 : sum = 150, average = 15
Done.
```

Использование нескольких параметров типа

Допускается создание шаблонов с несколькими параметрами типа. Предположим, что требуется класс, содержащий два вида значений. Для этой цели можно создать шаблонный класс Pair. (Между прочим, STL содержит подобный шаблон, который

называется pair.) В листинге 14.19 приведен небольшой пример. В нем методы first() const и second() const выводят хранимые значения, а методы first() и second(), благодаря возврату ссылок на данные-члены класса Pair, позволяют присвоить хранимым величинам новые значения.

Листинг 14.19. pairs.cpp

```
// pairs.cpp -- определение и использование шаблона Pair
#include <iostream>
#include <string>
template <class T1, class T2>
class Pair
{
private:
   T1 a;
   T2 b;
public:
   T1 & first();
   T2 & second();
   T1 first() const { return a; }
   T2 second() const { return b; }
   Pair(const T1 & aval, const T2 & bval) : a(aval), b(bval) { }
   Pair() {}
};
template<class T1, class T2>
T1 & Pair<T1,T2>::first()
{
   return a;
}
template<class T1, class T2>
T2 & Pair<T1,T2>::second()
{
   return b;
}
int main()
{
   using std::cout;
   using std::endl;
   using std::string;
   Pair<string, int> ratings[4] =
   {
      Pair<string, int>("The Purpled Duck", 5),
      Pair<string, int>("Jaquie's Frisco Al Fresco", 4),
      Pair<string, int>("Cafe Souffle", 5),
      Pair<string, int>("Bertie's Eats", 3)
   };
   int joints = sizeof(ratings) / sizeof (Pair<string, int>);
   cout << "Rating:\t Eatery\n";            // вывод рейтингов закусочных
   for (int i = 0; i < joints; i++)
      cout << ratings[i].second() << ":\t "
           << ratings[i].first() << endl;
   cout << "Oops! Revised rating:\n";       // вывод пересмотренного рейтинга
   ratings[3].first() = "Bertie's Fab Eats";
   ratings[3].second() = 6;
   cout << ratings[3].second() << ":\t "
        << ratings[3].first() << endl;
   return 0;
}
```

Обратите внимание, что в листинге 14.19 в функции `main()` для вызова конструкторов и в качестве аргумента для `sizeof` необходимо выражение `Pair<string,int>` — поскольку именем класса является `Pair<string,int>`, а не `Pair`.

А `Pair<char *,double>` представляет собой имя совершенно другого класса. Вывод программы из листинга 14.19 имеет следующий вид:

```
Rating: Eatery
5:      The Purpled Duck
4:      Jaquie's Frisco Al Fresco
5:      Cafe Souffle
3:      Bertie's Eats
Oops! Revised rating:
6:      Bertie's Fab Eats
```

Параметры типа по умолчанию в шаблонах

Еще одно новое свойство шаблонных классов — возможность указания значений по умолчанию для параметров типа:

```
template <class T1, class T2 = int> class Topo {...};
```

В этом случае компилятор использует `int` в качестве типа `T2`, если значение для `T2` отсутствует:

```
Topo<double, double> m1;      // тип T1 — double, тип T2 — double
Topo<double> m2;              // тип T1 — double, тип T2 — int
```

Это свойство часто используется в STL (см. главу 16), если типом по умолчанию является класс.

Хотя можно задать значения по умолчанию для типов параметров шаблонных классов, для параметров шаблонных функций это сделать нельзя. Тем не менее, значения по умолчанию нетипизированных параметров можно указывать как для шаблонных классов, так и для шаблонных функций.

Специализации шаблона

Для шаблонов классов, как и шаблонов функций, возможны неявные создания экземпляров, явные создания экземпляров и явные специализации, которые все вместе также называются *специализациями*. Шаблон описывает класс через обобщенный тип, а специализация — это объявление класса, сгенерированное для конкретного типа.

Неявное создание экземпляров

В примерах шаблонов, которые вы видели до сих пор в этой главе, используется *неявное создание экземпляров*. При этом объявление одного или более объектов задает нужный тип, и компилятор генерирует на основе общего шаблона специализированное определение класса:

```
ArrayTP<int, 100> stuff;      // неявное создание экземпляра
```

Компилятор не создает неявное создание экземпляра класса, пока не потребуется его объект:

```
ArrayTP<double, 30> * pt;     // указатель, пока еще объекты не нужны
pt = new ArrayTP<double, 30>; // теперь объект нужен
```

Второй оператор заставляет компилятор сгенерировать определение класса, а также объект, созданный согласно этому определению.

Явное создание экземпляров

Компилятор обеспечивает *явное создание экземпляра* объявления класса, если класс объявлен с применением ключевого слова `template`, а также указан необходимый тип или типы. Объявление класса должно находиться в том же пространстве имен, что и определение шаблона. Например, следующая строка кода объявляет, что `ArrayTP<string, 100>` является классом:

```
template class ArrayTP<string, 100>; //генерирует класс ArrayTP<string, 100>
```

В этом случае компилятор генерирует определение класса, включая определения методов, даже если не создаются или упоминаются объекты класса. Как и в случае неявного создания экземпляров, руководством для генерирования специализации служит общий шаблон.

Явная специализация

Явная специализация – это определение конкретного типа (или типов), который должен использоваться вместо общего шаблона. Иногда необходимо изменить шаблон так, чтобы он вел себя по-разному при создании экземпляров для различных типов – в этом случае можно создать явную специализацию. Предположим, например, что определен шаблон класса, представляющий отсортированный массив, элементы которого сортируются непосредственно при занесении в массив:

```
template <class T>
class SortedArray
{
   ... // подробности не показаны
};
```

Предположим также, что для сравнения значений шаблон использует операцию >. Она хорошо работает для чисел, а также в случаях, когда `T` является типом класса, в котором определен метод `T::operator>()`. Однако такой способ не сработает, если `T` является строкой, представляемой с помощью типа `const char *`. Вообще говоря, шаблон будет работать, но строки окажутся отсортированными не по алфавиту, а по адресам. Поэтому требуется определение класса, где вместо операции > используется сравнение `strcmp()`. В этом случае можно указать явную специализацию шаблона – т.е. шаблон, определенный для одного конкретного типа, а не общего типа. Если запросу специализации удовлетворяет и специализированный шаблон, и общий шаблон, компилятор использует специализированный вариант.

Определение специализированного шаблона класса имеет вид:

```
template <> class ИмяКласса<имя-специализированного-типа> { ... };
```

Некоторые старые компиляторы могут распознавать только ранние формы без префикса `template <>`:

```
class ИмяКласса<имя-специализированного-типа> { ... };
```

Для создания шаблона `SortedArray`, специализированного для типа `const char *`, в современной нотации нужен примерно такой код:

```
template <> class SortedArray<const char *>
{
  ... // подробности не показаны
};
```

Здесь для сравнения значений массива код реализации должен использовать вместо операции > функцию strcmp(). Теперь запросы шаблона SortedArray для типа const char * будут применять специализированное определение вместо более общего определения шаблона:

```
SortedArray<int> scores;        // используется общее определение
SortedArray<const char *> dates; // используется специализированное определение
```

Частичная специализация

В C++ разрешена *частичная специализация*, которая частично ограничивает общность шаблона. Например, используя частичную специализацию, можно задать конкретный тип для одного из параметров типа:

```
// Общий шаблон
template <class T1, class T2> class Pair {...};

// Специализация, в которой для T2 указан тип int
template <class T1> class Pair<T1, int> {...};
```

Угловые скобки <>, следующие за ключевым словом template, объявляют параметры типов, которые пока еще не специализированы. таким образом, второе объявление указывает для T2 тип int, но оставляет параметр T1 открытым. Обратите внимание, что указание всех типов приводит к пустым угловым скобкам и получению завершенной явной специализации:

```
// Специализация, в которой для T1 и T2 указан тип int
template <> class Pair<int, int> {...};
```

Если у компилятора есть выбор, он применяет наиболее специальный шаблон. Вот что произойдет для трех приведенных выше шаблонов:

```
Pair<double, double> p1; // используется общий шаблон Pair
Pair<double, int> p2;    // используется частичная специализация Pair<T1, int>
Pair<int, int> p3;       // используется явная специализация Pair<int, int>
```

Можно частично специализировать существующий шаблон, введя специальную версию для указателей:

```
template<class T>        // общая версия
class Feeb { ... };
template<class T*>       // частичная специализация с указателем
class Feeb { ... };      // измененный код
```

Если предоставить тип, который не является указателем, компилятор задействует общую версию, а если использовать указатель, компилятор выберет специализацию с указателем:

```
Feeb<char> fb1;          // используется общий шаблон Feeb (T - это char)
Feeb<char *> fb2;        // используется специализация Feeb T* (T — это char)
```

Без частичной специализации для второго объявления будет выбран общий шаблон, интерпретирующий T как тип char *. А при частичной специализации будет выбран специализированный шаблон, интерпретирующий T как тип char.

Частичная специализация позволяет задавать различные ограничения. Например, пусть имеются следующие объявления:

```
// Общий шаблон
template <class T1, class T2, class T3> class Trio{...};
// Специализация, когда для T3 указан T2
template <class T1, class T2> class Trio<T1, T2, T2> {...};
```

```
// Специализация, когда для T3 и T2 указан T1*
template <class T1> class Trio<T1, T1*, T1*> {...};
```

Для этих объявлений компилятор выберет следующие варианты:

```
Trio<int, short, char *> t1;        // используется общий шаблон
Trio<int, short> t2;                // используется Trio<T1, T2, T2>
Trio<char, char *, char *> t3;      // используется Trio<T1, T1*, T1*>
```

Шаблоны-члены

Шаблоны могут быть членами структуры, класса или шаблонного класса. Эти свойства необходимы библиотеке STL для полного определения своей структуры. В листинге 14.20 приведен небольшой пример шаблонного класса с вложенными в виде членов шаблонным классом и шаблонной функцией.

Листинг 14.20. tempmemb.cpp

```
// tempmemb.cpp — шаблоны-члены
#include <iostream>
using std::cout;
using std::endl;
template <typename T>
class beta
{
private:
   template <typename V>              // вложенный шаблонный класс-член
   class hold
   {
   private:
      V val;
   public:
      hold(V v = 0) : val(v) {}
      void show() const { cout << val << endl; }
      V Value() const { return val; }
   };
   hold<T> q;                         // шаблонный объект
   hold<int> n;                       // шаблонный объект
public:
   beta( T t, int i) : q(t), n(i) {}
   template<typename U>               // шаблонный метод
   U blab(U u, T t) { return (n.Value() + q.Value()) * u / t; }
   void Show() const { q.show(); n.show();}
};
int main()
{
   beta<double> guy(3.5, 3);
   cout << "T was set to double\n";   // T установлен в double
   guy.Show();
   cout << "V was set to T, which is double, then V was set to int\n";
        // V установлен в T, который double, затем V установлен в int
   cout << guy.blab(10, 2.3) << endl;
   cout << "U was set to int\n";      // U установлен в int
   cout << guy.blab(10.0, 2.3) << endl;
   cout << "U was set to double\n";   // U установлен в double
   cout << "Done\n";
   return 0;
}
```

Шаблон hold объявлен в закрытом разделе, поэтому он доступен только в пределах класса beta. Класс beta использует шаблон hold для определения двух членов данных:

```
hold<T> q;          // шаблонный объект
hold<int> n;        // шаблонный объект
```

n — объект hold, основанный на типе int, а член q — объект hold, основанный на типе T (параметр шаблона beta). Следующее объявление в функции main() присваивает T тип double, а q — тип hold<double>:

```
beta<double> guy(3.5, 3);
```

В методе blab() один тип (U) определен неявно, с помощью значения аргумента при вызове метода, а другой тип (T) определен типом создания экземпляра объекта. В данном примере объявление для guy назначает T тип double. Первый аргумент при вызове метода в следующем операторе назначает U тип int, соответствующий значению 10:

```
cout << guy.blab(10, 2.5) << endl;
```

Таким образом, хотя автоматическое преобразование типов из-за смешения типов приводит к вычислению blab() как double, возвращаемое значение, имеющее тип U, должно быть int. Поэтому оно усекается до 28, как показано в выводе программы:

```
T was set to double
3.5
3
V was set to T, which is double, then V was set to int
28
U was set to int
28.2609
U was set to double
Done
```

Если в вызове guy.blab() заменить 10 на 10.0, то тип U будет установлен в double, поэтому типом возврата будет double, о чем говорит наличие 28.2609 в выводе.

Как упоминалось ранее, тип второго параметра в определении объекта guy устанавливается в double. Но в отличие от первого параметра, тип второго параметра не задается вызовом функции. Например, показанный ниже оператор по-прежнему реализует blah() как blah(int, double), и значение 3 будет преобразовано в тип double по обычным правилам соответствия прототипам функций:

```
cout << guy.blab(10, 3) << endl;
```

Можно объявить класс hold и метод blah в шаблоне beta и определить их за пределами этого шаблона. Правда, некоторые старые компиляторы вообще не воспринимают шаблоны-члены, а другие допускают их в том виде, который представлен в листинге 14.20, но не разрешают определять вне класса. Однако при наличии современного компилятора можно определить шаблонные методы за пределами шаблона beta следующим образом:

```
template <typename T>
class beta
{
private:
   template <typename V>  // объявление
```

```
    class hold;
    hold<T> q;
    hold<int> n;
public:
    beta(T t, int i) : q(t), n(i) {}
    template<typename U>   // объявление
    U blab(U u, T t);
    void Show() const { q.show(); n.show(); }
};

// Определение члена
template <typename T>
  template<typename V>
    class beta<T>::hold
    {
    private:
       V val;
    public:
       hold(V v = 0) : val(v) {}
       void show() const { std::cout << val << std::endl; }
       V Value() const { return val; }
    };

// Определение члена
template <typename T>
  template <typename U>
    U beta<T>::blab(U u, T t)
    {
       return (n.Value() + q.Value()) * u / t;
    }
```

Определения должны идентифицировать `T`, `V` и `U` как параметры шаблона. Из-за вложенности шаблонов необходимо использовать синтаксис

```
template <typename T>
  template <typename V>
```

а не синтаксис

```
template<typename T, typename V>
```

`hold` и `blab` в определениях должны быть заданы как члены класса `beta<T>`, и для этого применяется операция разрешения контекста.

Шаблоны как параметры

Мы уже видели, что шаблоны могут иметь параметры типа, такие как `typename T`, и нетипизированные параметры вроде `int n`. Шаблоны также могут иметь параметры, которые сами являются шаблонами. Это еще одно дополнение, связанное с шаблонами, которое использовалось для реализации STL.

В листинге 14.21 показан пример, который начинается со следующих строк:

```
template <template <typename T> class Thing>
class Crab
```

Здесь `template <typename T> class Thing` — параметр-шаблон, причем `template <typename T> class` — тип, а `Thing` — параметр. Что это означает? Предположим, имеется объявление

```
Crab<King> legs;
```

Чтобы оно работало, аргумент шаблона King должен быть шаблонным классом, а его объявление должно соответствовать объявлению параметра-шаблона Thing:

```
template <typename T>
class King {...};
```

Класс Crab в листинге 14.21 объявляет два объекта:

```
Thing<int> s1;
Thing<double> s2;
```

Предыдущее объявление для legs привело бы к подстановке King<int> вместо Thing<int> и King<double> вместо Thing<double>. Однако в листинге 14.21 приведено следующее объявление:

```
Crab<Stack> nebula;
```

Поэтому в данном случае Thing<int> реализуется как Stack<int>, а Thing<double> — как Stack<double>. В общем, параметр шаблона Thing заменяется любым шаблонным типом, используемым в качестве аргумента шаблона в объявлении объекта Crab.

Объявление класса Crab основано на трех предположениях о шаблонном классе, представленном параметром Thing. Этот класс должен содержать методы push() и pop(), а эти методы должны иметь определенный интерфейс. Класс Crab может использовать любой шаблонный класс, который соответствует типу Thing и содержит методы push() и pop(). В этой главе рассматривается один такой класс — шаблон Stack, определенный в stacktp.h. Этот класс и применяется в примере.

Листинг 14.21. tempparm.cpp

```
// tempparm.cpp -- шаблоны как параметры
#include <iostream>
#include "stacktp.h"
template <template <typename T> class Thing>
class Crab
{
private:
   Thing<int> s1;
   Thing<double> s2;
public:
   Crab() {};
   // Предполагается, что класс thing имеет члены push() и pop()
   bool push(int a, double x) { return s1.push(a) && s2.push(x); }
   bool pop(int & a, double & x){ return s1.pop(a) && s2.pop(x); }
};
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   Crab<Stack> nebula;
   // Stack должен соответствовать шаблону template <typename T> class Thing
   int ni;
   double nb;
   cout << "Enter int double pairs, such as 4 3.5 (0 0 to end):\n";
        // Ввод пар чисел int и double
   while (cin>> ni >> nb && ni > 0 && nb > 0)
   {
      if (!nebula.push(ni, nb))
          break;
   }
```

```
    while (nebula.pop(ni, nb))
        cout << ni << ", " << nb << endl;
    cout << "Done.\n";
    return 0;
}
```

Ниже показан пример запуска программы из листинга 14.21:

```
Enter int double pairs, such as 4 3.5 (0 0 to end):
50 22.48
25 33.87
60 19.12
0 0
60, 19.12
25, 33.87
50, 22.48
Done.
```

Шаблонные параметры допускается смешивать с обычными параметрами. Например, объявление класса `Crab` может начинаться так:

```
template <template <typename T> class Thing, typename U, typename V>
class Crab
{
private:
    Thing<U> s1;
    Thing<V> s2;
    ...
```

Сейчас типы, сохраняемые в членах `s1` и `s2`, являются обобщенными, а не жестко закодированными типами. Поэтому в программе потребуется изменить определение `nebula` следующим образом:

```
Crab<Stack, int, double> nebula;  // T=Stack, U=int, V=double
```

Шаблонный параметр `T` является шаблонным типом, а параметры типов `U` и `V` — нешаблонными типами.

Шаблонные классы и друзья

У объявлений шаблонных классов также могут быть друзья, которые принадлежат одной из трех перечисленных ниже категорий.

- Нешаблонные друзья.
- Связанные шаблонные друзья — тип друга определяется типом класса при создании его экземпляра.
- Не связанные шаблонные друзья — все специализации друга являются друзьями для всех специализаций класса.

Рассмотрим пример для каждого случая.

Нешаблонные дружественные функции для шаблонных классов

Объявим в шаблонном классе обычную функцию в качестве друга:

```
template <class T>
class HasFriend
{
public:
  friend void counts(); // дружественная для всех созданий экземпляров HasFriend
    ...
};
```

Здесь функция counts() объявляется дружественной для всех возможных созданий экземпляров шаблона. Например, она будет дружественной для класса HasFriend<int> и для класса HasFriend<string>.

Функция counts() не вызывается объектом (она является дружественной, а не функцией-членом) и она не принимает каких-либо объектных параметров. Каким же образом она обращается к объекту HasFriend? Существует несколько вариантов. Она может иметь доступ к глобальному объекту; она может иметь доступ к локальным объектам через глобальный указатель; она может создать собственные объекты; и она может иметь доступ к статическим членам-данным, расположенным отдельно от объекта.

Предположим, что для дружественной функции требуется создать аргумент типа шаблонного класса. Возможно ли, например, следующее объявление друга?

```
friend void report(HasFriend &);              // возможно ли?
```

Ответ — невозможно. Дело в том, что объект HasFriend не может существовать. Существуют только конкретные специализации, такие как HasFriend<short>. Для создания аргумента типа шаблонного класса необходимо указать специализацию, например:

```
template <class T>
class HasFriend
{
   friend void report(HasFriend<T> &);        // связанный друг шаблона
   ...
};
```

Чтобы понять, что здесь происходит, представьте себе специализацию, генерируемую при объявлении объекта конкретного типа:

```
HasFriend<int> hf;
```

Компилятор заменит параметр шаблона T на int, и объявление друга примет следующий вид:

```
class HasFriend<int>
{
   friend void report(HasFriend<int> &);      // связанный друг шаблона
   ...
};
```

То есть функция report() с параметром HasFriend<int> становится дружественной для класса HasFriend<int>. Аналогично, функция report() с параметром HasFriend<double> будет перегруженной версией report(), которая является дружественной для класса HasFriend<double>.

Обратите внимание, что report() — нешаблонная функция: шаблоном является лишь ее параметр. Это означает, что для использования друзей необходимо определить явные специализации:

```
void report(HasFriend<short> &) { ... };      // явная специализация для short
void report(HasFriend<int> &) { ... };        // явная специализация для int
```

Эти моменты демонстрируются в листинге 14.22. В шаблоне HasFriend имеется статический член ct. Это означает, что любая конкретная специализация класса содержит собственный статический член. Метод counts(), являющийся другом для всех специализаций HasFriend, выводит значения ct из двух конкретных специализаций — HasFriend<int> и HasFriend<double>. В программе имеются также две функции reports(), каждая из которых является дружественной для одной конкретной специализации HasFriend.

Листинг 14.22. frnd2tmp.cpp

```
// frnd2tmp.cpp -- шаблонный класс с нешаблонными друзьями
#include <iostream>
using std::cout;
using std::endl;
template <typename T>
class HasFriend
{
private:
    T item;
    static int ct;
public:
    HasFriend(const T & i) : item(i) {ct++;}
    ~HasFriend() {ct--;}
    friend void counts();
    friend void reports(HasFriend<T> &);  // template parameter
};

// Каждая специализация имеет собственный статический член данных
template <typename T>
int HasFriend<T>::ct = 0;

// Нешаблонный друг для всех классов HasFriend<T>
void counts()
{
    cout << "int count: " << HasFriend<int>::ct << "; ";
    cout << "double count: " << HasFriend<double>::ct << endl;
}

// Нешаблонный друг для класса HasFriend<int>
void reports(HasFriend<int> & hf)
{
    cout <<"HasFriend<int>: " << hf.item << endl;
}

// Нешаблонный друг для класса HasFriend<double>
void reports(HasFriend<double> & hf)
{
    cout <<"HasFriend<double>: " << hf.item << endl;
}

int main()
{
    cout << "No objects declared: ";   // объекты пока не объявлены
    counts();

    HasFriend<int> hfi1(10);
    cout << "After hfi1 declared: ";   // после объявления hfi1
    counts();

    HasFriend<int> hfi2(20);
    cout << "After hfi2 declared: ";   // после объявления hfi2
    counts();

    HasFriend<double> hfdb(10.5);
    cout << "After hfdb declared: ";   // после объявления hfdb
    counts();
    reports(hfi1);
    reports(hfi2);
    reports(hfdb);
    return 0;
}
```

Некоторые компиляторы могут выдать предупреждение об использовании нешаблонной дружественной функции. Ниже показан вывод программы из листинга 14.22:

```
No objects declared: int count: 0; double count: 0
After hfi1 declared: int count: 1; double count: 0
After hfi2 declared: int count: 2; double count: 0
After hfdb declared: int count: 2; double count: 1
HasFriend<int>: 10
HasFriend<int>: 20
HasFriend<double>: 10.5
```

Связанные шаблонные функции, дружественные шаблонным классам

Рассмотренный выше пример можно изменить, сделав дружественные функции также шаблонами. В частности, можно создать связанных друзей шаблона — чтобы каждая специализация класса получала соответствующую специализацию друга. Этот прием немного сложнее, чем в случае нешаблонных друзей, и состоит из трех шагов.

На первом шаге перед определением класса необходимо объявить каждую шаблонную функцию:

```
template <typename T> void counts();
template <typename T> void report(T &);
```

Затем внутри функции нужно снова объявить шаблоны в качестве друзей. Вот операторы, которые объявляют специализации, основанные на типе параметра шаблонного класса:

```
template <typename TT>
class HasFriendT
{
   ...
   friend void counts<TT>();
   friend void report<>(HasFriendT<TT> &);
};
```

Угловые скобки <> в объявлениях означают специализации шаблона. В случае `report()` скобки <> могут быть пустыми, т.к. аргумент типа шаблона можно получить из аргумента функции

```
HasFriendT<TT>
```

Но возможен и такой вариант:

```
report< HasFriendT<TT> >(HasFriendT<TT> &)
```

Функция `counts()` не имеет параметров, поэтому для определения ее специализации нужно задействовать аргумент шаблона (`<TT>`). Обратите внимание, что `TT` — тип параметра для класса `HasFriendT`. Чтобы понять эти объявления, лучше представить, чем они станут при объявлении объекта конкретной специализации. Предположим, например, что объявлен такой объект:

```
HasFriendT<int> squack;
```

Компилятор подставит вместо `TT` тип `int` и сгенерирует следующее определение класса:

```
class HasFriendT<int>
{
   ...
   friend void counts<int>();
   friend void report<>(HasFriendT<int> &);
};
```

Одна специализация основана на типе TT, который преобразуется в int, а другая — на HasFriendT<TT>, который преобразуется в HasFriendT<int>. Таким образом, специализации шаблона counts<int>() и report<HasFriendT<int> >() объявлены как друзья класса HasFriendT<int>.

Третье требование, которому должна удовлетворять программа — она должна содержать определения шаблонов для друзей. Все эти три аспекта демонстрируются в листинге 14.23. Обратите внимание, что в листинге 14.22 одна функция counts() является другом для всех классов HasFriend. А в листинге 14.23 имеются две функции counts(), но лишь одна из них является другом каждому созданному типу класса. Поскольку вызовы функции counts() не содержат параметров, из которых компилятор мог бы вывести требуемую специализацию, в этих вызовах используются формы count<int>() и count<double>(). Но в вызовах reports() компилятор может определять специализацию на основе типа аргумента. С тем же результатом можно использовать и форму <>:

```
report<HasFriendT<int> >(hfi2);  // эквивалентно report(hfi2);
```

Листинг 14.23. tmp2tmp.cpp

```
// tmp2tmp.cpp -- шаблонные друзья для шаблонного класса
#include <iostream>
using std::cout;
using std::endl;

// Прототипы шаблонов
template <typename T> void counts();
template <typename T> void report(T &);

// Шаблонный класс
template <typename TT>
class HasFriendT
{
private:
    TT item;
    static int ct;
public:
    HasFriendT(const TT & i) : item(i) {ct++;}
    ~HasFriendT() { ct--; }
    friend void counts<TT>();
    friend void report<>(HasFriendT<TT> &);
};

template <typename T>
int HasFriendT<T>::ct = 0;

// Определения дружественных функций для шаблона
template <typename T>
void counts()
{
   cout << "template size: " << sizeof(HasFriendT<T>) << "; ";  // размер шаблона
   cout << "template counts(): " << HasFriendT<T>::ct << endl;  // counts() из шаблона
}

template <typename T>
void report(T & hf)
{
    cout << hf.item << endl;
}
```

```
int main()
{
   counts<int>();
   HasFriendT<int> hfi1(10);
   HasFriendT<int> hfi2(20);
   HasFriendT<double> hfdb(10.5);
   report(hfi1);                          // генерирует report(HasFriendT<int> &)
   report(hfi2);                          // генерирует report(HasFriendT<int> &)
   report(hfdb);                          // генерирует report(HasFriendT<double> &)
   cout << "counts<int>() output:\n";     // вывод из counts<int>()
   counts<int>();
   cout << "counts<double>() output:\n"; // вывод из counts<double>()
   counts<double>();
   return 0;
}
```

Вывод программы из листинга 14.23 выглядит следующим образом:

```
template size: 4; template counts(): 0
10
20
10.5
counts<int>() output:
template size: 4; template counts(): 2
counts<double>() output:
template size: 8; template counts(): 1
```

Как видите, `counts<double>` сообщает размер шаблона, отличный от выводимого `counts<int>` — т.е. каждому типу `T` соответствует собственная дружественная функция `count()`.

Не связанные шаблонные функции, дружественные шаблонным классам

В предыдущем разделе связанные шаблонные дружественные функции являются специализациями для шаблона, определенного вне класса. Специализация `int` класса дает специализацию функции `int` и т.д. Объявив шаблон внутри класса, можно создать не связанные дружественные функции, когда любая специализация функции будет дружественной для любой специализации класса. У не связанных друзей параметры типа для шаблонов друзей отличаются от параметров типа для шаблонных классов:

```
template <typename T>
class ManyFriend
{
   ...
   template <typename C, typename D> friend void show2(C &, D &);
};
```

В листинге 14.24 приведен пример применения не связанных друзей. В этом примере вызов `show2(hfi1, hfi2)` соответствует следующей специализации:

```
void show2<ManyFriend<int> &, ManyFriend<int> &>
        (ManyFriend<int> & c, ManyFriend<int> & d);
```

Поскольку данная функция является другом для всех специализаций `ManyFriend`, она имеет доступ к членам `item` всех специализаций. Однако она использует доступ только к объектам `ManyFriend<int>`.

Аналогично, вызов `show2(hfd, hfi2)` соответствует такой специализации:

```
void show2<ManyFriend<double> &, ManyFriend<int> &>
        (ManyFriend<double> & c, ManyFriend<int> & d);
```

Эта функции также является другом для всех специализаций ManyFriend и использует доступ к члену item объекта ManyFriend<int>, а также к члену item объекта ManyFriend<double>.

Листинг 14.24. manyfrnd.cpp

```
// manyfrnd.cpp — не связанная шаблонная функция, дружественная шаблонному классу
#include <iostream>
using std::cout;
using std::endl;

template <typename T>
class ManyFriend
{
private:
   T item;
public:
   ManyFriend(const T & i) : item(i) {}
   template <typename C, typename D> friend void show2(C &, D &);
};

template <typename C, typename D> void show2(C & c, D & d)
{
   cout << c.item << ", " << d.item << endl;
}

int main()
{
   ManyFriend<int> hfi1(10);
   ManyFriend<int> hfi2(20);
   ManyFriend<double> hfdb(10.5);
   cout << "hfi1, hfi2: ";
   show2(hfi1, hfi2);
   cout << "hfdb, hfi2: ";
   show2(hfdb, hfi2);
   return 0;
}
```

Ниже показан вывод программы из листинга 14.24:

```
hfi1, hfi2: 10, 20
hfdb, hfi2: 10.5, 20
```

Псевдонимы шаблонов (C++11)

Бывает удобно, особенно при построении шаблонов, создавать псевдонимы для типов. Конструкция typedef позволяет создавать псевдонимы для специализаций шаблонов:

```
// Определение трех псевдонимов с помощью typedef
typedef std::array<double, 12> arrd;
typedef std::array<int, 12> arri;
typedef std::array<std::string, 12> arrst;
arrd gallons;            // gallons имеет тип std::array<double, 12>
arri days;               // days имеет тип std::array<int, 12>
arrst months;            // months имеет тип std::array<std::string, 12>
```

Но если приходится постоянно писать код, содержащий такие описания typedef, вы можете подумать, а не забыли ли вы какую-то языковую возможность, которая упрощает эту задачу, или не забыли ли добавить такую возможность в язык его разработ-

чики. В C++11, наконец, появилась ранее недоступная возможность — способ использовать шаблон для получения семейства псевдонимов. Вот как это выглядит:

```
template<typename T>
   using arrtype = std::array<T,12>;    // шаблон для создания
                                        // нескольких псевдонимов
```

Ниже объявлен псевдоним шаблона `arrtype`, который можно применять вместо спецификатора типа:

```
arrtype<double> gallons;      // gallons имеет тип std::array<double, 12>
arrtype<int> days;            // days имеет тип std::array<int, 12>
arrtype<std::string> months;  // months имеет тип std::array<std::string, 12>
```

Короче говоря, `arrtype<T>` означает тип `std::array<T,12>`.

В C++11 синтаксис `using =` можно использовать и не для шаблонов. В таких случаях он эквивалентен `typedef`:

```
typedef const char * pc1;              // синтаксис typedef
using pc2 = const char *;              // синтаксис using =
typedef const int *(*pa1)[10];         // синтаксис typedef
using pa2 = const int *(*)[10];        // синтаксис using =
```

По мере привыкания, эта новая форма окажется более понятной, т.к. она более четко отделяет имя типа от информации об этом типе.

В C++11 появилось еще одно дополнение — *шаблон с переменным числом аргументов* (variadic template), который позволяет определить шаблонный класс или шаблонную функцию с переменным количеством инициализаторов. Эта тема рассматривается в главе 18.

Резюме

В C++ имеется несколько средств для повторного использования кода. Общедоступное наследование, рассмотренное в главе 13, позволяет моделировать отношение *является*, когда производные классы могут повторно использовать код базовых классов. Закрытое и защищенное наследование также позволяет повторно использовать код базовых классов, но в этом случае моделируется отношение *содержит*. При закрытом наследовании открытые и защищенные члены базового класса становятся закрытыми членами производного класса. При защищенном наследовании открытые и защищенные члены базового класса становятся защищенными членами производного класса. То есть в обоих случаях открытый интерфейс базового класса становится внутренним интерфейсом для производного класса. Иногда это называют наследованием реализации, а не интерфейса, т.к. производный объект не может явно использовать интерфейс базового класса. Поэтому производный объект нельзя считать разновидностью базового объекта. А из-за этого указатель или ссылку на базовый объект нельзя применять для ссылки на объект производного класса без явного приведения типа.

Еще один способ повторного использования кода класса — разработка класса, члены которого сами являются объектами. Этот подход называется *включением*, *иерархическим представлением* или *композицией* и также моделирует отношение *содержит*. Включение проще реализовать и применять, чем закрытое или защищенное наследование, и поэтому оно используется чаще. Однако возможности закрытого и защищенного наследования слегка различаются. Например, наследование позволяет производному классу обращаться к защищенным членам базового класса. Оно также позволяет производному классу переопределять виртуальные функции, унаследованные от базо-

вого класса. Включение не является разновидностью наследования и поэтому не обеспечивает таких возможностей. Зато включение удобнее, если нужно создать несколько объектов одного класса. Например, класс `Country` (Страна) может содержать массив объектов `State` (Штат).

Множественное наследование позволяет использовать в классе код нескольких других классов. Закрытое и защищенное множественное наследование приводит к созданию отношений *содержит*, а открытое множественное наследование — к созданию отношений *является*. Применение множественного наследования приводит к возникновению проблем, связанных с неоднозначностью имен и неоднозначным наследованием базового класса. Для разрешения неоднозначности имен можно использовать квалификаторы класса, а для преодоления неоднозначности наследования — виртуальные базовые классы. Однако виртуальные базовые классы вводят новые правила для списка инициализации в конструкторах и для разрешения неоднозначности.

Шаблоны классов позволяют создать общую структуру класса, в которой тип (как правило, тип члена) представлен параметром типа. Обычно шаблон выглядит следующим образом:

```
template <class T>
class Ic
{
   T v;
   ...
public:
   Ic(const T & val) : v(val) { }
   ...
};
```

Здесь `T` — параметр типа, и он играет роль заполнителя для реального типа, который будет указан позднее. (Этот параметр может иметь любое допустимое в C++ имя, но обычно применяется `T` или `Type`.) В данном контексте слово `class` можно заменить словом `typename`:

```
template <typename T>     // эквивалентно template <class T>
class Rev {...} ;
```

Определение класса (создание экземпляра) генерируется при объявлении объекта класса и указании конкретного типа. Например, следующее объявление указывает компилятору сгенерировать объявление класса, в котором каждое вхождение парамет- ра типа `T` в шаблоне заменено типом `short`:

```
class Ic<short> sic;      // неявное создание экземпляра
```

В этом случае именем класса будет `Ic<short>`, а не `Ic`. Объявление `Ic<short>` называется *специализацией* шаблона. В данном случае это неявное создание экземпляра.

Явное создание экземпляра происходит при объявлении конкретной специализации класса с помощью ключевого слова `template`:

```
template class IC<int>;   // явное создание экземпляра
```

В этом случае компилятор использует общий шаблон для генерации специализации `Ic<int>`, даже если еще не затребован ни один объект этого класса.

Можно создать явную специализацию — специализированное определение класса, которое переопределяет определение шаблона. Для этого определение класса начинается с конструкции `template<>`, потом указывается имя шаблонного класса, а за ним — угловые скобки, содержащие тип требуемой специализации.

Например, можно создать специализированный класс `Ic` для указателей на символы:

```
template <> class Ic<char *>.
{
   char * str;
   ...
public:
   Ic(const char * s) : str(s) { }
   ...
};
```

Тогда объявление следующего вида будет использовать не общий шаблон, а специализированное определение для chic:

```
class Ic<char *> chic;
```

Шаблон класса может задавать несколько общих типов и может иметь нстипизированные параметры:

```
template <class T, class TT, int n>
class Pals {...};
```

Показанное ниже объявление сгенерирует неявное создание экземпляра, заменив T на double, TT на string и n на 6:

```
Pals<double, string, 6> mix;
```

Шаблон класса может иметь параметры, которые сами являются шаблонами:

```
template <template <typename T> class CL, typename U, int z>
class Trophy {...};
```

Здесь z — это значение типа int, U — имя типа, а CL — шаблон класса, опрсделснного конструкцией template <typename T>.

Шаблоны класса могут быть специализированными частично:

```
template <class T> Pals<T, T, 10> {...};
template <class T, class TT> Pals<T, TT, 100> {...};
template <class T, int n> Pals <T, T*, n> {...};
```

В первом примере создается специализация, в которой оба типа одинаковы, а n имеет значение 6. Во втором примере создается специализация для n, равного 100. В третьем примере создается специализация, в которой второй тип являстся указателем на первый тип.

Шаблонные классы могут быть членами других классов, структур и шаблонов.

Главная цель создания всех рассмотренных методов — предоставить программисту возможность повторного использования проверенного кода без его ручного копирования. Это упрощает программирование и повышает надежность программ.

Вопросы для самоконтроля

1. Для каждого набора классов из столбца А укажите, какое наследование — общедоступное или закрытое — лучше подходит для столбца Б.

А	Б
class Bear (Медведь)	class PolarBear (Белый медведь)
class Kitchen (Кухня)	class Home (Дом)
class Person (Человек)	class Programmer (Программист)
class Person (Человек)	class HorseAndJockey (Лошадь и жокей)
class Person, class Automobile (Человек, Автомобиль)	class Drive (Поездка)

2. Пусть имеются следующие определения:

```
class Frabjous {
private:
    char fab[20];
public:
    Frabjous(const char * s = "C++") : fab(s) { }
    virtual void tell() { cout << fab; }
};
class Gloam {
private:
    int glip;
    Frabjous fb;
public:
    Gloam(int g = 0, const char * s = "C++");
    Gloam(int g, const Frabjous & f);
    void tell();
};
```

Напишите определения для трех методов класса `Gloam`, если функция `tell()` из класса `Gloam` выводит значения `glip` и `fb`.

3. Пусть имеются следующие определения:

```
class Frabjous {
private:
    char fab[20];
public:
    Frabjous(const char * s = "C++") : fab(s) { }
    virtual void tell() { cout << fab; }
};
class Gloam : private Frabjous{
private:
    int glip;
public:
    Gloam(int g = 0, const char * s = "C++");
    Gloam(int g, const Frabjous & f);
    void tell();
};
```

Напишите определения для трех методов класса `Gloam`, если функция `tell()` из класса `Gloam` выводит значения `glip` и `fb`.

4. Пусть имеется следующее определение, основанное на шаблоне `Stack` из листинга 14.13 и на классе `Worker` из листинга 14.10:

```
Stack<Worker *> sw;
```

Напишите объявление класса, который будет сгенерирован (только объявление, без встроенных методов).

5. Воспользуйтесь определениями шаблонов, рассмотренных в этой главе, чтобы определить:
   - массив объектов `string`;
   - стек массивов значений `double`;
   - массив стеков указателей на объекты `Worker`.

   Сколько определений шаблонов классов сгенерировано в листинге 14.18?

6. Объясните разницу между виртуальными и невиртуальными базовыми классами.

Упражнения по программированию

1. Класс Wine (Вино) содержит объект-член типа string (см. главу 4) для названия вина и объект Pair из объектов valarray<int> (рассматривались в этой главе). Первый член каждого объекта Pair содержит год сбора винограда, а второй член — количества бутылок с вином урожая этх лет. Например, первый объект valarray объекта Pair содержит годы 1888, 1992 и 1996, а второй объект valarray — количества бутылок: 24, 48 и 144. Хорошо бы, чтобы объект Wine содержал целочисленный член для хранения возраста вина в годах. Для упрощения кода могут быть полезными следующие объявления typedef:

```
typedef std::valarray<int> ArrayInt;
typedef Pair<ArrayInt, ArrayInt> PairArray;
```

Таким образом, тип PairArray представляет тип Pair<std::valarray<int>, std::valarray<int> >. Реализуйте класс Wine, используя включение. Этот класс должен иметь конструктор по умолчанию и, как минимум, следующие конструкторы:

```
// Инициализация label значением l, количество лет - y,
// годы урожая - yr[], количество бутылок - bot[]
Wine(const char * l, int y, const int yr[], const int bot[]);
// Инициализация label значением l, количество лет - y,
// создаются объекты массива размером y
Wine(const char * l, int y);
```

Класс Wine должен содержать метод GetBottles(), который для объекта Wine заданного возраста предлагает пользователю ввести соответствующие значения для года урожая и количества бутылок. Метод Label() должен возвращать ссылку на название вина, а метод sum() — общее количество бутылок во втором объекте valarray<int> из объекта Pair.

Программа должна предлагать пользователю ввести название вина, количество элементов в массиве, а также год и количество бутылок для каждого элемента массива. Программа должна использовать эти данные для создания объекта Wine и вывода информации, хранимой в объекте. Для справки ниже приведен пример тестовой программы:

```
// pe14-1.cpp — класс Wine с использованием включения
int main ( void )
{
   using std::cin;
   using std::cout;
   using std::endl;
   cout << "Enter name of wine: ";      // ввод названия вина
   char lab[50];
   cin.getline(lab, 50);
   cout << "Enter number of years: ";   // ввод количества годов сбора винограда
   int yrs;
   cin >> yrs;
   Wine holding(lab, yrs);         // сохранение названия, лет,
                                   // создание массивов из yrs элементов
   holding.GetBottles();           // предложение ввести год и количество бутылок
   holding.Show();                 // вывод содержимого объекта
   const int YRS = 3;
   int y[YRS] = {1993, 1995, 1998};
   int b[YRS] = { 48, 60, 72};
```

```
    // Создание нового объекта, инициализация
    // с использованием данных из массивов y и b
    Wine more("Gushing Grape Red",YRS, y, b);
    more.Show();
    cout << "Total bottles for " << more.Label()  // используется метод Label()
         << ": " << more.sum() << endl;           // используется метод sum()
    cout << "Bye\n";
    return 0;
}
```

А так может выглядеть вывод программы:

```
Enter name of wine: Gully Wash
Enter number of years: 4
Enter Gully Wash data for 4 year(s):
Enter year: 1988
Enter bottles for that year: 42
Enter year: 1994
Enter bottles for that year: 58
Enter year: 1998
Enter bottles for that year: 122
Enter year: 2001
Enter bottles for that year: 144
Wine: Gully Wash
       Year   Bottles
       1988   42
       1994   58
       1998   122
       2001   144
Wine: Gushing Grape Red
       Year   Bottles
       1993   48
       1995   60
       1998   72
Total bottles for Gushing Grape Red: 180
Bye
```

2. Выполните еще раз упражнение 1, но вместо включения используйте закрытое наследование. Здесь также могут пригодиться несколько объявлений `typedef`. Подумайте, как можно применить следующие операторы:

```
PairArray::operator=(PairArray(ArrayInt(), ArrayInt()));
cout << (const string &)(*this);
```

Полученный класс должен работать с тестовой программой, приведенной в упражнении 1.

3. Определите шаблон `QueueTp`. Протестируйте его, создав очередь указателей на `Worker` (см. листинг 14.10), и примените его в программе, такой как приведенная в листинге 14.12.

4. Класс `Person` (Человек) предназначен для хранения имени и фамилии человека. Кроме конструкторов он содержит метод `Show()` для вывода этих данных. Класс `Gunslinger` (Снайпер) виртуально порожден от класса `Person`. Он содержит член `Draw()`, который возвращает значение типа `double` — время, необходимое снайперу для перехода в боевую готовность. Класс также имеет член типа `int`, содержащий количество насечек на винтовке. И, наконец, класс содержит функцию `Show()`, которая выводит всю эту информацию.

Класс PokerPlayer (Игрок в покер) виртуально порожден от класса Person. Он имеет метод Draw(), который возвращает случайное число в диапазоне от 1 до 52 — значение карты. (Можно создать класс Card с членами, определяющими масть и рубашку карты, чтобы метод Draw() возвращал значение типа Card.) Класс PokerPlayer использует функцию Show() класса Person. Класс BadDude (Хулиган) открыто порожден от классов Gunslinger и PokerPlayer. Он содержит член Gdraw(), возвращающий время вынимания оружия, и член Cdraw(), возвращающий следующую вытянутую карту. У него есть соответствующая функция Show(). Определите все упомянутые классы и методы вместе с другими необходимыми методами (такими как методы для задания значений объекта) и протестируйте их с помощью простой программы, подобной представленной в листинге 14.12.

5. Ниже приведено несколько объявлений классов:

```
// emp.h -- заголовочный файл для класса abstr_emp и его дочерних классов
#include <iostream>
#include <string>
class abstr_emp
{
private:
   std::string fname;            // имя abstr_emp
   std::string lname;            // фамилия abstr_emp
   std::string job;
public:
   abstr_emp();
   abstr_emp(const std::string & fn, const std::string & ln,
             const std::string & j);
   virtual void ShowAll() const; // выводит все данные с именами
   virtual void SetAll();        // запрашивает значения у пользователя
   friend std::ostream &
      operator<<(std::ostream & os, const abstr_emp & e);
   // Выводит только имя и фамилию
   virtual ~abstr_emp() = 0;     // виртуальный базовый класс
};
class employee : public abstr_emp
{
public:
   employee();
   employee(const std::string & fn, const std::string & ln,
            const std::string & j);
   virtual void ShowAll() const;
   virtual void SetAll();
};
class manager: virtual public abstr_emp
{
private:
   int inchargeof;               // количество управляемых abstr_emp
protected:
   int InChargeOf() const { return inchargeof; }       // вывод
   int & InChargeOf(){ return inchargeof; }            // ввод
public:
   manager();
   manager(const std::string & fn, const std::string & ln,
           const std::string & j, int ico = 0);
```

```
    manager(const abstr_emp & e, int ico);
    manager(const manager & m);
    virtual void ShowAll() const;
    virtual void SetAll();
};
class fink: virtual public abstr_emp
{
private:
    std::string reportsto;                    // кому выводить отчеты
protected:
    const std::string ReportsTo() const { return reportsto; }
    std::string & ReportsTo() { return reportsto; }
public:
    fink();
    fink(const std::string & fn, const std::string & ln,
         const std::string & j, const std::string & rpo);
    fink(const abstr_emp & e, const std::string & rpo);
    fink(const fink & e);
    virtual void ShowAll() const;
    virtual void SetAll();
};
class highfink: public manager, public fink  // надзор за управляющими
{
public:
    highfink();
    highfink(const std::string & fn, const std::string & ln,
             const std::string & j, const std::string & rpo,
             int ico);
    highfink(const abstr_emp & e, const std::string & rpo, int ico);
    highfink(const fink & f, int ico);
    highfink(const manager & m, const std::string & rpo);
    highfink(const highfink & h);
    virtual void ShowAll() const;
    virtual void SetAll();
};
```

Здесь в иерархии классов используется множественное наследование с виртуальным базовым классом. Поэтому не забывайте о специальных правилах для списков инициализации в конструкторах. Обратите также внимание на наличие нескольких методов с защищенным доступом. Это упрощает код некоторых методов highfink. (Например, если метод `highfink::ShowAll()` просто вызывает `fink::ShowAll()` и `manager::ShowAll()`, то это приводит к двукратному вызову `abstr_emp::ShowAll()`.) Реализуйте эти методы и протестируйте классы. Ниже приведена минимальная тестовая программа:

```
// pe14-5.cpp
// useemp1.cpp -- использование классов abstr_emp
#include <iostream>
using namespace std;
#include "emp.h"
int main(void)
{
    employee em("Trip", "Harris", "Thumper");
    cout << em << endl;
    em.ShowAll();
    manager ma("Amorphia", "Spindragon", "Nuancer", 5);
```

```
    cout << ma << endl;
    ma.ShowAll();
    fink fi("Matt", "Oggs", "Oiler", "Juno Barr");
    cout << fi << endl;
    fi.ShowAll();
    highfink hf(ma, "Curly Kew");                // укомплектовано?
    hf.ShowAll();
    cout << "Press a key for next phase:\n";
          // Нажать любую клавишу для следующей фазы
    cin.get();
    highfink hf2;
    hf2.SetAll();
    cout << "Using an abstr_emp * pointer:\n";
          // Использование указателя abstr_emp *
    abstr_emp * tri[4] = {&em, &fi, &hf, &hf2};
    for (int i = 0; i < 4; i++)
       tri[i]->ShowAll();
    return 0;
}
```

Почему не определена операция присваивания?

Почему методы `ShowAll()` и `SetAll()` виртуальные?

Почему класс `abstr_emp` является виртуальным базовым классом?

Почему в классе `highfink` нет раздела данных?

Почему достаточно только одной версии операции `operator<<()`?

Что произойдет, если код в конце программы модифицировать следующим образом:

```
abstr_emp tri[4] = {em, fi, hf, hf2};
for (int i = 0; i < 4; i++)
   tri[i].ShowAll();
```

15

Друзья, исключения и многое другое

В ЭТОЙ ГЛАВЕ...

- Дружественные классы
- Дружественные методы классов
- Вложенные классы
- Генерация исключений и блоки `try` и `catch`
- Классы исключений
- Динамическая идентификация типов (RTTI)
- Операции `dynamic_cast` и `typeid`
- Операции `static_cast`, `const_cast` и `reinterpret_cast`

В этой главе рассматриваются некоторые тонкие моменты и несколько последних расширений языка C++. К тонким моментам относятся дружественные классы, дружественные функции-члены и вложенные классы (т.е. классы, определенные внутри других классов). А из последних расширений здесь будут рассмотрены исключения, идентификация типов во время выполнения (runtime type identification – RTTI) и улучшенное управление приведением типов. Исключения в C++ предоставляют механизм обработки нестандартных ситуаций, которые иначе приводят к останову программы. RTTI представляет собой механизм определения типов объектов. Новые операции приведения типов повышают надежность таких приведений. Последние три возможности появились в C++ относительно недавно, и старыми компиляторами они не поддерживаются.

Друзья

В некоторых примерах этой книги использовались дружественные функции как часть расширенного интерфейса классов. Такие функции – не единственный вид друзей, которые может иметь класс. Другом может быть и какой-нибудь класс. В этом случае все методы дружественного класса имеют доступ к закрытым и защищенным методам исходного класса. Но в качестве дружественных функций класса можно указать и лишь отдельные функции-члены другого класса. Класс сам определяет, какие функции, функции-члены или классы являются для него друзьями; дружественные отношения нельзя навязать извне. Таким образом, хотя друзья разрешают доступ к закрытой части класса, они не противоречат духу объектно-ориентированного программирования – наоборот, они придают дополнительную гибкость открытому интерфейсу.

Дружественные классы

Когда возникает необходимость сделать один класс дружественным другому классу? Рассмотрим пример. Предположим, что нужно написать программный эмулятор телевизора и пульта дистанционного управления для него. Для этого естественно определить класс `Tv`, представляющий телевизор, и класс `Remote`, представляющий пульт дистанционного управления. Понятно, что между этими классами должна существовать какая-то взаимосвязь, но какая? Пульт – это не телевизор, и наоборот, поэтому отношение *является*, возникающее при открытом наследовании, не годится. Ни один, ни другой класс не являются компонентом другого, поэтому отношение *содержит*, возникающее при включении, закрытом и защищенном наследовании, тоже не подходит. Но пульт позволяет изменять состояние телевизора, и это наводит на мысль сделать класс `Remote` дружественным классу `Tv`.

Давайте определим класс `Tv`. Можно представить телевизор как набор переменных-членов, описывающих состояние телевизора. Вот список возможных состояний телевизора:

- включен или выключен;
- переключение на заданный канал;
- настройка уровня громкости;
- режим настройки кабеля или антенны;
- сигнал от антенны или видеоплеера.

Режим настройки отражает тот факт, что в США частотное расстояние между каналами, начиная с 14 канала, различается для кабельного и эфирного телевидения. Выбор входа позволяет переключить телевизор на прием кабельного или эфирного телевидения либо сигнала от DVD-плеера. Некоторые устройства имеют дополнительные возможности — например, несколько входов от DVD/Blu-ray, но для нашего примера достаточно приведенного списка.

Телевизор также обладает и другими параметрами, которые нельзя представить переменными состояния. Например, телевизоры отличаются по количеству каналов, которые они могут принимать, и можно создать член, описывающий это свойство.

Теперь необходимо создать класс с методами для изменения состояний телевизора. У многих современных телевизоров органы управления скрыты за декоративными панелями, но они все-таки позволяют менять настройки без дистанционного управления. Правда, при этом обычно можно лишь переключаться на соседние каналы, но нельзя выбрать канал произвольно. Для изменения громкости существуют аналогичные кнопки — одна для увеличения, другая для уменьшения громкости.

Пульт дистанционного управления должен дублировать органы управления телевизора. Многие из его методов могут быть реализованы с помощью методов класса `Tv`. Но, кроме того, пульт дистанционного управления позволяет переключиться на произвольный канал. То есть можно сразу перейти со 2-го на 20-й канал, не перебирая все промежуточные. Многие пульты дистанционного управления могут работать в двух и более режимах, например, управление телевизором и управление DVD-плеером.

На основе этих рассуждений можно написать определения, приведенные в листинге 15.1. В них имеются несколько констант, определенных как перечисления. Следующий оператор делает `Remote` дружественным классом:

```
friend class Remote;
```

Объявление дружественной конструкции может находиться в открытом, закрытом или защищенном разделе — место не имеет значения. Поскольку класс `Remote` ссылается на класс `Tv`, компилятор должен знать о классе `Tv` до начала обработки класса `Remote`. Самый простой способ — определить класс `Tv` первым. Либо можно использовать объявление `forward`; мы рассмотрим эту возможность позже.

Листинг 15.1. `tv.h`

```
// tv.h -- классы Tv и Remote
#ifndef TV_H_
#define TV_H_

class Tv
{
public:
    friend class Remote;  // Remote имеет доступ к закрытой части Tv
    enum {Off, On};
    enum {MinVal,MaxVal = 20};
    enum {Antenna, Cable};
    enum {TV, DVD};

    Tv(int s = Off, int mc = 125) : state(s), volume(5),
       maxchannel(mc), channel(2), mode(Cable), input(TV) {}
    void onoff() {state = (state == On)? Off : On;}
    bool ison() const {return state == On;}
    bool volup();
    bool voldown();
    void chanup();
    void chandown();
```

```
    void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
    void set_input() {input = (input == TV)? DVD : TV;}
    void settings() const;      // отображение всех настроек
private:
    int state;                  // On или Off
    int volume;                 // дискретные уровни громкости
    int maxchannel;             // максимальное количество каналов
    int channel;                // текущий канал
    int mode;                   // эфирное или кабельное телевидение
    int input;                  // TV или DVD
};

class Remote
{
private:
    int mode;                   // управление TV или DVD
public:
    Remote(int m = Tv::TV) : mode(m) {}
    bool volup(Tv & t) {return t.volup();}
    bool voldown(Tv & t) {return t.voldown();}
    void onoff(Tv & t) {t.onoff(); }
    void chanup(Tv & t) {t.chanup();}
    void chandown(Tv & t) {t.chandown();}
    void set_chan(Tv & t, int c) {t.channel = c;}
    void set_mode(Tv & t) {t.set_mode();}
    void set_input(Tv & t) {t.set_input();}
};

#endif
```

Большинство методов в листинге 15.1 определено встроенным образом. Обратите внимание, что каждый метод класса `Remote`, отличный от конструктора, принимает в качестве параметра ссылку на объект `Tv` — т.е. пульт дистанционного управления должен быть нацелен на конкретный телевизор. В листинге 15.2 приведены остальные определения. Функции управления громкостью изменяют уровень звука на единицу, пока он не достигнет максимального или минимального значения. Функции выбора канала используют циклический возврат: за минимальным значением канала, равным `1`, сразу следует максимальный канал, равный `maxchannel`, и наоборот.

Многие методы для переключения между двумя состояниями используют условную операцию:

```
void onoff() {state = (state == On)? Off : On;}
```

Поскольку значениями `state` могут быть только `true` и `false` — или эквивалентные им `0` и `1` — это действие можно записать более компактно с помощью операции исключающего "ИЛИ", объединенной с операцией присваивания (`^=`; см. приложение Д):

```
void onoff() {state ^= 1;}
```

На самом деле в переменной `unsigned char` можно хранить до восьми двоичных состояний и переключать их по отдельности. Но это совсем другая история, и для нее нужны побитовые операции, которые также описаны в приложении Д.

Листинг 15.2. tv.cpp

```
// tv.cpp — методы для класса Tv (методы Remote являются встроенными)
#include <iostream>
#include "tv.h"
```

```
bool Tv::volup()
{
   if (volume < MaxVal)
   {
      volume++;
      return true;
   }
   else
      return false;
}
bool Tv::voldown()
{
   if (volume > MinVal)
   {
      volume--;
      return true;
   }
   else
      return false;
}
void Tv::chanup()
{
   if (channel < maxchannel)
      channel++;
   else
      channel = 1;
}
void Tv::chandown()
{
   if (channel > 1)
      channel--;
   else
      channel = maxchannel;
}
void Tv::settings() const
{
   using std::cout;
   using std::endl;
   cout << "TV is " << (state == Off? "Off" : "On") << endl;  // выключен или включен
   if (state == On)
   {
     cout << "Volume setting = " << volume << endl;           // уровень громкости
     cout << "Channel setting = " << channel << endl;         // номер канала
     cout << "Mode = "
          << (mode == Antenna? "antenna" : "cable") << endl;  // антенна или кабель
     cout << "Input = "
          << (input == TV? "TV" : "DVD") << endl;             // вход: TV или DVD
   }
}
```

В листинге 15.3 приведена короткая программа для тестирования некоторых возможностей. Один и тот же пульт используется для управления двумя телевизорами.

Листинг 15.3. use_tv.cpp

```
// use_tv.cpp -- использование классов Tv и Remote
#include <iostream>
#include "tv.h"
int main()
{
   using std::cout;
   Tv s42;
   cout << "Initial settings for 42\" TV:\n";    // начальные настройки телевизора 42
   s42.settings();
   s42.onoff();
   s42.chanup();
   cout << "\nAdjusted settings for 42\" TV:\n";     // отрегулированные настройки
                                                     // телевизора 42
   s42.settings();
   Remote grey;
   grey.set_chan(s42, 10);
   grey.volup(s42);
   grey.volup(s42);
   cout << "\n42\" settings after using remote:\n"; // настройки телевизора 42 после
                                                     // использования пульта
   s42.settings();
   Tv s58(Tv::On);
   s58.set_mode();
   grey.set_chan(s58,28);
   cout << "\n58\" settings:\n";                    // настройки телевизора 58
   s58.settings();
   return 0;
}
```

Ниже показан вывод программы из листингов 15.1, 15.2 и 15.3:

```
Initial settings for 42" TV:
TV is Off

Adjusted settings for 42" TV:
TV is On
Volume setting = 5
Channel setting = 3
Mode = cable
Input = TV

42" settings after using remote:
TV is On
Volume setting = 7
Channel setting = 10
Mode = cable
Input = TV

58" settings:
TV is On
Volume setting = 5
Channel setting = 28
Mode = antenna
Input = TV
```

Главный вывод из этого упражнения: дружественность классов является естественным понятием, которое позволяет выразить некоторые отношения. Без какой-то формы дружбы пришлось бы сделать закрытые разделы класса Tv открытыми или создать один громоздкий класс, охватывающий и телевизор, и пульт дистанционного управления. Но такое решение все равно не позволило бы выразить тот факт, что один пульт можно использовать с несколькими телевизорами.

Дружественные функции-члены

Просматривая последний пример, можно заметить, что большинство методов класса Remote реализовано с использованием открытого интерфейса класса Tv. Это значит, что для этих методов дружественный статус и не нужен. В самом деле, единственный метод класса Remote, который непосредственно обращается к закрытому члену класса Tv – это Remote::set_chan(), поэтому другом должен быть только он. Можно сделать дружественными лишь отдельные члены класса, а не весь класс целиком, однако это менее удобно. Понадобится следить за порядком различных определений и объявлений. Посмотрим, почему это происходит.

Сделать метод Remote::set_chan() дружественным классу Tv можно, объявив его в качестве друга в объявлении класса Tv:

```
class Tv
{
   friend void Remote::set_chan(Tv & t, int c);
   ...
};
```

Однако для обработки этого оператора компилятор уже должен просмотреть определение класса Remote. Иначе он не будет знать, что Remote является классом, а set_chan() – методом этого класса. Это наводит на мысль поместить определение Remote перед определением Tv. Но методы класса Remote ссылаются на объекты Tv, а это значит, что определение Tv должно находиться перед определением Remote. Избежать циклических ссылок позволяет *упреждающее (предварительное) объявление*. Для этого перед определением Remote необходимо вставить следующий оператор:

```
class Tv;        // упреждающее объявление
```

В результате получится вот что:

```
class Tv;        // упреждающее объявление
class Remote { ... };
class Tv { ... };
```

А можно ли вместо этого применить такой порядок?

```
class Remote;    // упреждающее объявление
class Tv { ... };
class Remote { ... };
```

Оказывается, что нельзя. Причина в том, что, как было сказано, когда компилятор видит, что метод класса Remote объявлен как дружественный в объявлении класса Tv, он должен уже просмотреть объявление всего класса Remote и, в частности, метода set_chan().

Есть и другая сложность. В листинге 15.1 объявление Remote содержит следующий встроенный код:

```
void onoff(Tv & t) { t.onoff(); }
```

Здесь вызывается метод Tv, и поэтому компилятор должен заранее просмотреть объявление класса Tv, чтобы знать, какие методы он содержит. Но, как мы видели, за этим объявлением должно следовать объявление Remote. Решением в данном случае может быть *объявление* методов Remote до определения класса Tv и их непосредственные *определения* после класса Tv. Это приводит к такой последовательности:

```
class Tv;              // упреждающее объявление
class Remote { ... }; // методы, использующие Tv, в виде только прототипов
class Tv { ... };
// Определения методов Remote
```

Прототипы Remote выглядят следующим образом:

```
void onoff(Tv & t);
```

Для обработки этого прототипа компилятору нужно лишь знать, что Tv является классом, и упреждающее объявление предоставляет ему эту информацию. Когда компилятор дойдет до непосредственных определений методов, объявление класса Tv им уже прочитано, и он будет располагать информацией, необходимой для компиляции этих методов. Ключевое слово inline в определениях методов также позволяет сделать методы встроенными. В листинге 15.4 показан модифицированный заголовочный файл.

Листинг 15.4. tvfm.h

```
// tvfm.h -- классы Tv и Remote и дружественная функция-член
#ifndef TVFM_H_
#define TVFM_H_
class Tv;                  // упреждающее объявление
class Remote
{
public:
   enum State{Off, On};
   enum {MinVal,MaxVal = 20};
   enum {Antenna, Cable};
   enum {TV, DVD};
private:
   int mode;
public:
   Remote(int m = TV) : mode(m) {}
   bool volup(Tv & t);   // только прототип
   bool voldown(Tv & t);
   void onoff(Tv & t) ;
   void chanup(Tv & t) ;
   void chandown(Tv & t) ;
   void set_mode(Tv & t) ;
   void set_input(Tv & t);
   void set_chan(Tv & t, int c);
};
class Tv
{
public:
   friend void Remote::set_chan(Tv & t, int c);
   enum State{Off, On};
   enum {MinVal,MaxVal = 20};
   enum {Antenna, Cable};
   enum {TV, DVD};
   Tv(int s = Off, int mc = 125) : state(s), volume(5),
      maxchannel(mc), channel(2), mode(Cable), input(TV) {}
```

```
    void onoff() {state = (state == On)? Off : On;}
    bool ison() const {return state == On;}
    bool volup();
    bool voldown();
    void chanup();
    void chandown();
    void set_mode() {mode = (mode == Antenna)? Cable : Antenna;}
    void set_input() {input = (input == TV)? DVD : TV;}
    void settings() const;
private:
    int state;
    int volume;
    int maxchannel;
    int channel;
    int mode;
    int input;
};
// Методы Remote как встроенные функции
inline bool Remote::volup(Tv & t) { return t.volup();}
inline bool Remote::voldown(Tv & t) { return t.voldown();}
inline void Remote::onoff(Tv & t) { t.onoff(); }
inline void Remote::chanup(Tv & t) {t.chanup();}
inline void Remote::chandown(Tv & t) {t.chandown();}
inline void Remote::set_mode(Tv & t) {t.set_mode();}
inline void Remote::set_input(Tv & t) {t.set_input();}
inline void Remote::set_chan(Tv & t, int c) {t.channel = c;}
#endif
```

Если в файлах `tv.cpp` и `use_tv.cpp` включить `tvfm.h` вместо `tv.h`, результирующая программа будет вести себя так же, как исходная. Разница лишь в том, что теперь дружественным по отношению к классу `Tv` будет только один метод класса `Remote`, а не все. Эта разница показана на рис. 15.1.

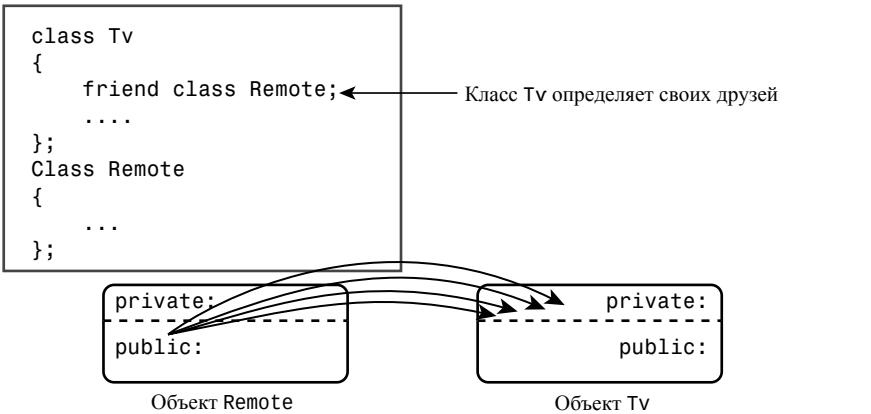
Вспомните, что встроенные функции имеют внутреннее связывание, т.е. определения функций должны находиться в том же файле, где они используются. Здесь встроенные определения находятся в заголовочном файле, поэтому при включении этого файла в файл, использующий определения, эти определения попадут в нужное место. Можно поместить определения и в файл реализации, но тогда нужно удалить ключевое слово `inline`, чтобы выполнялось внешнее связывание функций.

Кстати, если сделать дружественным весь класс `Remote`, упреждающее объявление станет ненужным, т.к. объявление друга само определяет `Remote` как класс.

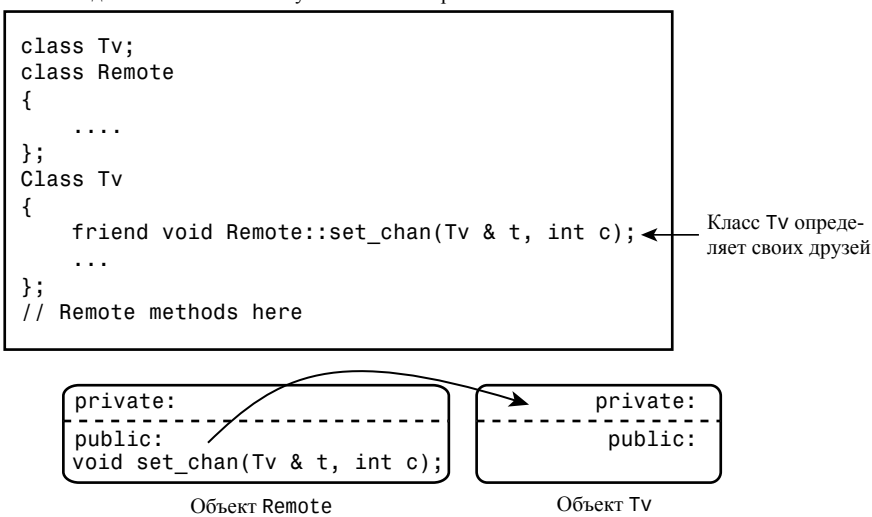
Другие дружественные отношения

Кроме уже рассмотренных в этой главе случаев возможны и другие сочетания друзей и классов. Рассмотрим вкратце некоторые из них.

Предположим, что у нас есть достижение передовых технологий — интерактивный пульт дистанционного управления. Он позволяет ввести ответ на вопрос, заданный в телевизионной программе, а телевизор может включить зуммер в пульте, если этот ответ неверен. Не вдаваясь в разные психологические тонкости, рассмотрим лишь аспекты программирования на C++. Такая система может использовать взаимные дружественные отношения: некоторые методы класса `Remote` могут, как и ранее, воздействовать на объект `Tv`, а некоторые методы класса `Tv` могут воздействовать на объект `Remote`. Этого можно добиться, сделав классы дружественными друг другу. То есть `Tv` будет другом для `Remote`, а `Remote` будет другом для `Tv`.



Все методы класса `Remote` могут влиять на закрытые члены класса `Tv`



Только метод `Remote::set_chan()` может влиять на закрытые члены класса `Tv`

***Рис. 15.1.** Дружественные классы и дружественные члены класса*

Здесь следует иметь в виду, что метод класса `Tv`, использующий объект `Remote`, может быть объявлен *до* объявления класса `Remote`, но должен быть определен *после* объявления, чтобы у компилятора была вся информация, необходимая для компиляции метода. Структура кода будет выглядеть следующим образом:

```
class Tv
{
friend class Remote;
public:
   void buzz(Remote & r);
   ...
};
class Remote
{
friend class Tv;
public:
   void Bool volup(Tv & t) { t.volup(); }
   ...
};
```

```
inline void Tv::buzz(Remote & r)
{
    ...
}
```

Поскольку объявление `Remote` расположено после объявления `Tv`, метод `Remote::volup()` можно определить в объявлении класса. Однако метод `Tv::buzz()` должен быть определен вне объявления класса `Tv`, чтобы это определение находилось после объявления класса `Remote`. Если нет необходимости вставлять встроенное определение `buzz()`, его нужно определить в отдельном файле с определениями методов.

Общие друзья

Другой пример использования друзей — когда функции нужен доступ к приватным данным двух разных классов. По идее такая функция должна быть функцией-членом каждого из этих классов, но это невозможно. Она может быть членом одного класса и другом другого. Но иногда лучше сделать ее дружественной обоим классам. Предположим, что имеется класс `Probe`, представляющий некий программируемый измерительный прибор, и класс `Analyzer`, представляющий программируемый анализатор. У каждого прибора есть внутренние часы, и их нужно синхронизировать. Можно использовать следующий код:

```
class Analyzer;  // упреждающее объявление
class Probe
{
    friend void sync(Analyzer & a, const Probe & p);  // синхронизация a с p
    friend void sync(Probe & p, const Analyzer & a);  // синхронизация p с a
    ...
};
class Analyzer
{
    friend void sync(Analyzer & a, const Probe & p);  // синхронизация a с p
    friend void sync(Probe & p, const Analyzer & a);  // синхронизация p с a
    ...
};

// Определение дружественных функций
inline void sync(Analyzer & a, const Probe & p)
{
    ...
}
inline void sync(Probe & p, const Analyzer & a)
{
    ...
}
```

Когда компилятор доходит до объявлений друзей в объявлении класса `Probe`, он, в силу наличия упреждающего объявления, уже знает, что `Analyzer` является классом.

Вложенные классы

C++ позволяет помещать объявление класса внутрь другого класса. Класс, объявленный внутри другого класса, называется *вложенным классом*. Вложенные классы ограничивают область видимости имен пределами класса и позволяют избежать конфликта имен. Функции-члены класса, содержащего вложенное объявление, могут создавать и использовать объекты вложенного класса. Извне взаимодействовать с вложенными

классами можно, только если они объявлены в открытой части класса и только с использованием операции разрешения контекста. (Старые версии C++ либо не поддерживают вложенные классы, либо поддерживают не полностью.)

Не путайте вложение классов и включение. Включение означает наличие объекта класса в качестве члена другого класса. Вложение не создает член класса, а определяет тип, который известен лишь локально в объемлющем классе.

Обычно вложенные классы создаются как вспомогательные при реализации другого класса, чтобы избежать конфликта имен. Вложенный класс уже был неявно представлен в классе `Queue` из листинга 12.10 (см. главу 12) как вложенное определение структуры:

```
class Queue
{
private:
   // Определения области действия класса
   // Node — это вложенное определение структуры, локальное для данного класса
   struct Node {Item item; struct Node * next;};
   ...
};
```

Поскольку структура — это класс, члены которого по умолчанию являются общедоступными, `Node` действительно представляет собой вложенный класс. Однако это определение не использует возможности вложенных классов. В частности, у него нет явного конструктора. Давайте восполним этот недостаток.

Сначала нужно найти, где в примере `Queue` создается объект `Node`. Просмотрев объявление класса (листинг 12.10) и определения методов (листинг 12.11), можно увидеть, что единственное место, где создаются объекты `Node` — это метод `enqueue()`:

```
bool Queue::enqueue(const Item & item)
{
   if (isfull())
      return false;
   Node * add = new Node;   // создание узла
   // В случае сбоя операция new генерирует исключение std::bad_alloc
   add->item = item;        // установка указателей на узлы
   add->next = NULL;
   ...
}
```

В этом коде после создания объекта `Node` его членам явно присваиваются значения. Такие действия уместнее выполнять в конструкторе.

Зная, где и как понадобится конструктор, можно написать соответствующее определение:

```
class Queue
{
   // Объявления с областью видимости класса
   // Node — вложенный класс, локальный по отношению к данному
   class Node
   {
   public:
      Item item;
      Node * next;
      Node(const Item & i) : item(i), next(0) { }
   };
   ...
};
```

В этом конструкторе член item инициализируется значением i, а указатель next устанавливается в 0; такая установка — один из способов создания нулевого указателя в C++. (Использование NULL требует включения в проект заголовочного файла с его определением. При наличии компилятора, совместимого с C++11, можно также написать nullptr.) Поскольку во всех узлах, создаваемых с помощью класса Queue, член next первоначально содержит нулевой указатель, то для этого класса достаточно одного конструктора. Теперь перепишем метод enqueue() с применением конструктора:

```
bool Queue::enqueue(const Item & item)
{
   if (isfull())
      return false;
   Node * add = new Node(item);   // создание и инициализация узла
   // В случае сбоя операция new генерирует исключение std::bad_alloc
   ...
}
```

Это сделает код enqueue() немного короче и надежнее, поскольку инициализация проводится автоматически, и программисту не нужно запоминать все необходимые действия.

В данном примере конструктор определен в объявлении класса. Если потребуется определить его в файле методов, то тогда определение должно учитывать, что класс Node определен внутри класса Queue. Это делается с помощью двух операций разрешения контекста:

```
Queue::Node::Node(const Item & i) : item(i), next(0) { }
```

Вложенные классы и доступ

С вложенными классами связаны два вида доступа. Во-первых, место объявления вложенного класса определяет его область видимости — т.е. указывает, какие части программы могут создавать объекты этого класса. Во-вторых, как и для любого другого класса, доступ к членам класса определяют открытый, закрытый и защищенный разделы класса. Место и способ использования вложенного класса зависят как от области видимости класса, так и от управления доступом. Рассмотрим эти положения.

Область видимости

Если вложенный класс определен в закрытом разделе другого класса, он известен только объемлющему классу. В последнем примере это относится к классу Node, вложенному в объявление класса Queue. Члены класса Queue могут использовать объекты Node и указатели на объекты Node, а другие части программы даже не подозревают о существовании класса Node. Если от Queue будет порожден другой класс, то Node для этого класса будет также невидим, поскольку производный класс не имеет прямого доступа к закрытой части базового класса.

Если вложенный класс объявлен в защищенном разделе другого класса, он будет видимым для этого класса, но не видимым извне. Поскольку вложенный класс известен во всем базовом классе, вовне его нужно использовать с квалификатором класса. Пусть имеется есть следующее объявление:

```
class Team                // Команда
{
public:
   class Coach { ... };   // Тренер
   ...
};
```

Теперь предположим, что есть безработный тренер, не имеющий команды. Для создания объекта Coach вне класса Team можно поступить следующим образом:

```
Team::Coach forhire;    // создание объекта Coach за пределами класса Team
```

Подобные рассуждения об области видимости применимы и к вложенным структурам и перечислимым типам. Многие программисты используют открытые перечислимые типы для задания констант класса, которые могут применяться в клиентских классах. Например, как уже было показано, многие реализации классов, созданные для поддержки средства iostream, используют эту технику для создания различных вариантов форматирования (более подробно об этом будет рассказано в главе 17).

В табл. 15.1 приведены свойства области видимости для вложенных классов, структур и перечислений.

Таблица 15.1. Свойства области видимости для вложенных классов, структур и перечислений

Где определен во вложенном классе	Доступен для вложенного класса	Доступен для класса, порожденного от вложенного класса	Доступен извне
Закрытый раздел	Да	Нет	Нет
Защищенный раздел	Да	Да	Нет
Открытый раздел	Да	Да	Да, с квалификатором класса

Управление доступом

Если класс оказывается в области видимости, вступают в действие правила управления доступом. Для вложенных классов действуют те же правила доступа, что и для обычных классов. Объявление класса Node внутри объявления класса Queue не дает классу Queue привилегированный доступ к Node, как и классу Node при доступе к Queue. То есть объект класса Queue может обращаться только к открытым членам объекта Node. По этой причине в примере с Queue все члены класса Node сделаны открытыми. Это идет вразрез с обычной практикой, когда члены данных создаются закрытыми, но класс Node не виден за пределами класса Queue, поскольку объявлен в его закрытой части. Значит, методы Queue могут непосредственно обращаться к членам Node, а клиенты, использующие класс Queue, не могут.

В общем, место объявления класса определяет область видимости этого класса. Если какой-то класс находится в области видимости, доступ программы к членам вложенного класса определяется обычными правилами доступа — открытый, закрытый, защищенный, дружественный.

Вложение в шаблонах

Мы уже видели, что шаблоны удобны для создания контейнерных классов, таких как Queue. Конечно, интересно, возникнут ли проблемы при преобразовании определения класса Queue в шаблон. Так вот, проблем не будет. В листинге 15.5 показано, каким образом можно выполнить такое преобразование. Как обычно для шаблонных классов, заголовочный файл содержит шаблон класса и функции-шаблоны методов.

Листинг 15.5. queuetp.h

```
// queuetp.h — шаблон очереди с вложенным классом
#ifndef QUEUETP_H_
```

```
#define QUEUETP_H_
template <class Item>
class QueueTP
{
private:
   enum {Q_SIZE = 10};
   // Node — определение вложенного класса
   class Node
   {
   public:
      Item item;
      Node * next;
      Node(const Item & i):item(i), next(0){ }
   };
   Node * front;                    // указатель на начало очереди
   Node * rear;                     // указатель на конец очереди
   int items;                       // текущее количество элементов в очереди
   const int qsize;                 // максимальное количество элементов в очереди
   QueueTP(const QueueTP & q) : qsize(0) {}
   QueueTP & operator=(const QueueTP & q) { return *this; }
public:
   QueueTP(int qs = Q_SIZE);
   ~QueueTP();
   bool isempty() const
   {
      return items == 0;
   }
   bool isfull() const
   {
      return items == qsize;
   }
   int queuecount() const
   {
      return items;
   }
   bool enqueue(const Item &item); // добавление item в конец
   bool dequeue(Item &item);       // удаление item из начала
};
// Методы QueueTP

template <class Item>
QueueTP<Item>::QueueTP(int qs) : qsize(qs)
{
   front = rear = 0;
   items = 0;
}
template <class Item>
QueueTP<Item>::~QueueTP()
{
   Node * temp;
   while (front != 0)               // пока очередь не пуста
   {
      temp = front;                 // сохранение адреса первого элемента
      front = front->next;          // сдвиг указателя на следующий элемент
      delete temp;                  // удаление предыдущего первого
   }
}
```

```
// Добавление элемента в очередь
template <class Item>
bool QueueTP<Item>::enqueue(const Item & item)
{
   if (isfull())
      return false;
   Node * add = new Node(item);     // создание узла
   // В случае сбоя операция new генерирует исключение std::bad_alloc
   items++;
   if (front == 0)                  // если очередь пуста,
      front = add;                  // элемент добавляется в начало
   else
      rear->next = add;             // иначе добавляем в конец
   rear = add;                      // последний элемент назначается новым узлом
   return true;
}
// Помещение первого элемента в переменную item и удаление его из очереди
template <class Item>
bool QueueTP<Item>::dequeue(Item & item)
{
   if (front == 0)
      return false;
   item = front->item;              // item — первый элемент в очереди
   items--;
   Node * temp = front;             // сохранение местоположения первого элемента
   front = front->next;             // сдвиг на следующий элемент
   delete temp;                     // удаление предыдущего первого элемента
   if (items == 0)
      rear = 0;
   return true;
}
#endif
```

Один интересный момент: класс `Node` определен в листинге 15.5 через общий тип `Item`. Поэтому при наличии объявления

```
QueueTp<double> dq;
```

`Node` будет содержать значения типа `double`, а объявление

```
QueueTp<char> cq;
```

приведет к тому, что `Node` будет хранить значения типа `char`. Эти два класса `Node` определены в двух раздельных классах `QueueTP`, поэтому конфликт имен не возникает, и один узел имеет тип `QueueTP<double>::Node`, а другой — тип `QueueTP<char>::Node`.

В листинге 15.6 приведена короткая программа для тестирования нового класса.

Листинг 15.6. `nested.cpp`

```
// nested.cpp -- использование очереди, имеющей вложенный класс
#include <iostream>
#include <string>
#include "queuetp.h"
int main()
{
   using std::string;
   using std::cin;
   using std::cout;
   QueueTP<string> cs(5);
```

```
    string temp;
    while(!cs.isfull())
    {
        cout << "Please enter your name. You will be "
                "served in the order of arrival.\n"
                "name: ";  // ввод имени и фамилии
        getline(cin, temp);
        cs.enqueue(temp);
    }
    cout << "The queue is full. Processing begins!\n";
          // Очередь полна; начало обслуживания
    while (!cs.isempty())
    {
        cs.dequeue(temp);
        cout << "Now processing " << temp << "...\n";
    }
    return 0;
}
```

Ниже приведен пример запуска программы из листингов 15.5 и 15.6:

```
Please enter your name. You will be served in the order of arrival.
name: Kinsey Millhone
Please enter your name. You will be served in the order of arrival.
name: Adam Dalgliesh
Please enter your name. You will be served in the order of arrival.
name: Andrew Dalziel
Please enter your name. You will be served in the order of arrival.
name: Kay Scarpetta
Please enter your name. You will be served in the order of arrival.
name: Richard Jury
The queue is full. Processing begins!
Now processing Kinsey Millhone...
Now processing Adam Dalgliesh...
Now processing Andrew Dalziel...
Now processing Kay Scarpetta...
Now processing Richard Jury...
```

Исключения

При выполнении программ иногда встречаются ситуации, препятствующие их нормальному продолжению. Например, программа может попытаться открыть недоступный файл, запросить памяти больше, чем доступно в данный момент, или столкнуться со значением, которое она не может обработать. Обычно программисты стараются предвидеть подобные события. Исключения в C++ предлагают мощный и гибкий механизм обработки таких ситуаций. Они представляют собой относительно недавнее расширение языка C++, поэтому некоторые старые компиляторы не поддерживают их. Кроме того, в отдельных компиляторах эта возможность отключена по умолчанию, и ее надо включить с помощью опций компилятора.

Перед тем как заняться исключениями, рассмотрим некоторые более простые методы, доступные программистам. Для примера рассмотрим функцию, которая рассчитывает среднее гармоническое значение. *Среднее гармоническое* двух чисел определяется как обратное значение среднего их обратных значений. Это определение можно выразить в виде выражения:

```
2.0 × x × y / (x + y)
```

Обратите внимание, что если y равно x с обратным знаком, вычисление по формуле приведет к делению на ноль — совершенно неприемлемая операция. Многие новейшие компиляторы выполняют деление на ноль, получая в результате специальное значение в формате с плавающей точкой, которое обозначает бесконечность. В cout это значение отображается как Inf, inf, INF или что-то в этом роде. Другие компиляторы генерируют программы, которые при выполнении деления на ноль завершаются аварийно. Ясно, что лучше создать код, который ведет себя одинаково в любой системе.

Вызов abort()

Один из способов выхода из создавшейся ситуации — вызов функции abort(), если один аргумент равен другому с обратным знаком. Прототип функции abort() находится в заголовочном файле cstdlib (или stdlib.h). В типичной реализации при ее вызове в стандартный поток ошибок (тот же самый, который используется объектом cerr) отправляется сообщение вроде "abnormal program termination" ("аварийное завершение программы"), и выполнение программы прекращается. Кроме того, операционной системе или родительскому процессу возвращается значение, зависящее от реализации и означающее аварийное завершение. Выводит ли функция abort() содержимое файловых буферов (области памяти, используемые для хранения данных при операциях с файлами) или нет, также зависит от реализации. Можно использовать функцию exit(), которая точно выводит содержимое буферов — правда, без вывода сообщения. В листинге 15.7 приведена короткая программа, использующая функцию abort().

Листинг 15.7. error1.cpp

```
//error1.cpp -- использование функции abort()
#include <iostream>
#include <cstdlib>
double hmean(double a, double b);
int main()
{
    double x, y, z;
    std::cout << "Enter two numbers: ";             // запрос на ввод двух чисел
    while (std::cin >> x >> y)
    {
        z = hmean(x,y);
        std::cout << "Harmonic mean of " << x << " and " << y
                  << " is " << z << std::endl;       // вывод среднего гармонического
        std::cout << "Enter next set of numbers <q to quit>: ";
                  // запрос следующих двух чисел
    }
    std::cout << "Bye!\n";
    return 0;
}
double hmean(double a, double b)
{
    if (a == -b)
    {
      std::cout << "untenable arguments to hmean()\n";  // неверные аргументы для hmean()
      std::abort();
    }
    return 2.0 * a * b / (a + b);
}
```

Ниже показан пример запуска программы из листинга 15.7:

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
untenable arguments to hmean()
abnormal program termination
```

Обратите внимание, что вызов функции `abort()` из `hmean()` сразу же прекращает выполнение программы без возврата в `main()`. Вообще говоря, разные компиляторы выдают разные аварийные сообщения. Вот еще одно такое сообщение:

```
This application has requested the Runtime to terminate it
in an unusual way. Please contact the application's support
team for more information.
```

Это приложение обратилось к среде времени выполнения с запросом на необычное завершение. Для получения дополнительной информации свяжитесь с группой поддержки приложения.

(Наверняка вам понравилось предположение о том, что такой небольшой программы есть группа поддержки.) Аварийного завершения программы можно избежать, проверяя значения `x` и `y` перед вызовом функции `hmean()`. Но не очень-то надежно надеяться, что программист достаточно знает (или беспокоится) о выполнении такой проверки.

Возврат кода ошибки

Для определения возникшей проблемы удобнее не просто прекращение выполнения программы, а использование значения, возвращаемого функцией. Например, член `get(void)` класса `ostream` обычно возвращает ASCII-код очередного введенного символа, однако в случае достижения конца файла он возвращает специальное значение `EOF`. Для `hmean()` этот подход не годится: любое числовое значение является допустимым возвращаемым значением, и не существует специального значения для индикации проблемы. В этой ситуации можно в качестве аргумента функции применять указатель или ссылку — это позволяет вернуть значение в вызывающую программу, и на основе этого значения определить успешность выполнения функции. Разновидность такого приема используется в семействе `istream` перегруженных операций >>. Информируя вызывающую программу об успехе или неудаче, можно предпринять действия, отличные от аварийного завершения программы. В листинге 15.8 приведен пример такого подхода. В нем функция `hmean()` определена по-другому: теперь она возвращает значение `bool`, которое показывает, успешно ли выполнена функция. В ней также добавлен третий аргумент для возврата ответа.

Листинг 15.8. `error2.cpp`

```
//error2.cpp -- возврат кода ошибки
#include <iostream>
#include <cfloat>   // (или float.h) для DBL_MAX
bool hmean(double a, double b, double * ans);
int main()
{
   double x, y, z;
   std::cout << "Enter two numbers: ";          // запрос на ввод двух чисел
   while (std::cin >> x >> y)
   {
```

```
        if (hmean(x,y,&z))
            std::cout << "Harmonic mean of " << x << " and " << y
                      << " is " << z << std::endl;  // вывод среднего гармонического
        else
            std::cout << "One value should not be the negative "
                 << "of the other - try again.\n";  // одно значение не может быть равно
                                                    // другому с обратным знаком
        std::cout << "Enter next set of numbers <q to quit>: ";
                  // запрос следующих двух чисел
    }
    std::cout << "Bye!\n";
    return 0;
}
bool hmean(double a, double b, double * ans)
{
    if (a == -b)
    {
        *ans = DBL_MAX;
        return false;
    }
    else
    {
        *ans = 2.0 * a * b / (a + b);
        return true;
    }
}
```

Ниже показан пример запуска программы из листинга 15.8:

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
One value should not be the negative of the other - try again.
Enter next set of numbers <q to quit>: 1 19
Harmonic mean of 1 and 19 is 1.9
Enter next set of numbers <q to quit>: q
Bye!
```

Замечания по программе

Структура программы в листинге 15.8 позволяет пользователю продолжить работу, обойдя ситуацию, возникшую из-за неправильного ввода. Конечно, программа возлагает на пользователя обработку возвращаемых функцией значений — программисты это делают не всегда. Например, чтобы сохранять примеры как можно более краткими, в большинстве листингов этой книги не проверяется, успешно ли завершен вывод в поток `cout`.

В качестве третьего аргумента можно использовать указатель или ссылку. Многие программисты предпочитают применять указатели на аргументы встроенных типов — тогда понятно, какой аргумент использовался для ответа.

Еще один вариант сохранения возвращаемых значений — применение глобальных переменных. Функция, в которой возможны проблемы, в аварийных ситуациях может заносить в глобальную переменную определенное значение, а вызывающая программа может проверить эту переменную. Этот метод реализован в стандартной математической библиотеке языка C, в которой для подобных целей служит глобальная переменная `errno`. Разумеется, нужно обеспечить, чтобы какая-нибудь другая функция не использовала глобальную переменную с таким же именем для других целей.

Механизм исключений

А теперь посмотрим, как позволяет справляться с проблемами механизм исключений. В языке C++ *исключение* — это реакция на нештатную ситуацию, возникшую во время выполнения программы, например, при делении на ноль. Исключения позволяют передать управление из одной части программы в другую.

Для управления исключениями доступны три компонента:

- генерация исключения;
- перехват исключения обработчиком;
- использование блока `try`.

Программа генерирует исключение, когда возникает проблемная ситуация. Например, можно изменить функцию `hmean()` из листинга 15.7, чтобы она генерировала исключение вместо вызова функции `abort()`. В сущности, оператор `throw` является оператором перехода, поскольку при этом управление передается операторам в другом месте программы. Ключевое слово `throw` является признаком генерации исключения. После него указывается значение — например, символьная строка или объект, — обозначающее природу исключения.

Программа перехватывает исключение с помощью *обработчика исключений*, расположенного в том месте программы, где исключение необходимо обработать. Ключевое слово `catch` означает перехват исключения. Обработчик исключения начинается с ключевого слова `catch`, за которым следует объявление типа (в круглых скобках), представляющее тип исключения, которому оно соответствует. За ними в фигурных скобках располагается блок кода, выполняющего необходимые действия. Ключевое слово `catch` вместе с типом исключения играет роль метки, определяющей точку в программе, куда должно быть передано управление при возникновении исключения. Обработчик исключения называется также *блоком `catch`* или *блоком перехвата*.

Блок `try` представляет собой блок кода, в котором активизируются определенные исключения. За ним следуют один или несколько блоков `catch`. Блок `try` начинается с ключевого слова `try`, а за ним в фигурных скобках находится код, в котором отслеживаются исключения.

Проще всего продемонстрировать взаимодействие этих трех элементов на коротком примере, приведенном в листинге 15.9.

Листинг 15.9. `error3.cpp`

```
// error3.cpp — использование исключения
#include <iostream>
double hmean(double a, double b);
int main()
{
   double x, y, z;
   std::cout << "Enter two numbers: ";              // запрос на ввод двух чисел
   while (std::cin >> x >> y)
   {
     try {                                         // начало блока try
        z = hmean(x,y);
     }                                             // конец блока try
     catch (const char * s)                        // начало обработчика исключений
     {
       std::cout << s << std::endl;
       std::cout << "Enter a new pair of numbers: "; //запрос на ввод новой пары чисел
       continue;
     }                                             // конец обработчика исключений
```

```
        std::cout << "Harmonic mean of " << x << " and " << y
                  << " is " << z << std::endl;        // вывод среднего гармонического
        std::cout << "Enter next set of numbers <q to quit>: ";
                  // запрос следующих двух чисел
    }
    std::cout << "Bye!\n";
    return 0;
}
double hmean(double a, double b)
{
    if (a == -b)
        throw "bad hmean() arguments: a = -b not allowed";
    return 2.0 * a * b / (a + b);
}
```

Ниже показан пример запуска программы из листинга 15.9:

```
Enter two numbers: 3 6
Harmonic mean of 3 and 6 is 4
Enter next set of numbers <q to quit>: 10 -10
bad hmean() arguments: a = -b not allowed
Enter a new pair of numbers: 1 19
Harmonic mean of 1 and 19 is 1.9
Enter next set of numbers <q to quit>: q
Bye!
```

Замечания по программе

Блок `try` в листинге 15.9 выглядит следующим образом:

```
try {                              // начало блока try
    z = hmean(x,y);
}                                  // конец блока try
```

Если какой-то оператор в этом блоке приведет к генерации исключения, то обработка исключения произойдет в блоках `catch`, следующих за этим блоком `try`. Если программа вызовет `hmean()` где-нибудь вне этого (или любого другого) блока `try`, она не сможет обработать исключение.

Генерация исключения выглядит так:

```
if (a == -b)
    throw "bad hmean() arguments: a = -b not allowed";
```

В данном случае генерируемое исключение представляет собой строку `"bad hmean() arguments: a = -b not allowed"`. Исключение может иметь строковый тип, как в данном примере, или любой другой тип C++. Обычно исключение имеет тип класса, как будет показано ниже в этой главе.

Выполнение оператора `throw` слегка похоже на выполнение оператора возврата в функции — в том смысле, что он завершает выполнение функции. Однако вместо возврата управления вызывающей программе оператор `throw` заставляет программу возвращаться по текущей цепочке вызовов функций до тех пор, пока не будет найден блок `try`. В листинге 15.9 такой функцией является вызывающая функция. Ниже мы рассмотрим пример с возвратом более чем на один уровень. В данном случае оператор `throw` передает управление обратно в `main()`. Здесь программа ищет обработчик исключения (следующий за блоком `try`), который соответствует типу сгенерированного исключения.

Обработчик, или блок `catch`, выглядит следующим образом:

```
catch (const char * s)                          // начало обработчика исключений
{
  std::cout << s << std::endl;
  std::cout << "Enter a new pair of numbers: "; // запрос на ввод новой пары чисел
    continue;
}                                               // конец обработчика исключений
```

Блок `catch` немного похож на определение функции, но это не функция. Ключевое слово `catch` указывает, что это обработчик, а выражение `char * s` означает, что обработчик соответствует строковым исключениям. Такое объявление `s` аналогично определению аргумента функции, и если возникшее исключение соответствует этому объявлению, оно присваивается `s`, а затем программа выполняет код внутри фигурных скобок.

Если программа выполнила все операторы внутри блока `try` без возникновения исключений, она пропускает все блоки `catch` и переходит к выполнению операторов, следующих за обработчиками исключений. Поэтому когда программа из листинга 15.9 обработает значения `3` и `6`, она переходит сразу к оператору вывода и выводит результат.

Давайте проследим, что происходит в примере, после того как значения `10` и `-10` будут обработаны функцией `hmean()`. Проверка `if` заставляет `hmean()` сгенерировать исключение. Выполнение `hmean()` прекращается. Просматривая стек вызовов, программа определяет, что функция `hmean()` была вызвана внутри блока `try` в `main()`. Затем программа ищет блок `catch` с типом, который соответствует типу исключения. Единственный существующий блок `catch` имеет параметр `char *`, поэтому он соответствует исключению. Найдя соответствие, программа присваивает переменной `s` значение `"bad hmean() arguments: a = -b not allowed"`. Затем программа выполняет код обработчика. Сначала выводится строка `s` с описанием исключения. После этого программа предлагает ввести новые данные. И, наконец, выполняется оператор `continue`, который пропускает остаток тела цикла `while` и передает управление его началу. То, что оператор `continue` переносит управление в начало цикла, говорит о том, что обработчик является частью цикла, и что строка `catch` ведет себя как метка, направляющая поток выполнения программы (рис. 15.2).

А что произойдет, если функция сгенерирует исключение, но нет ни одного блока `try` или соответствующего обработчика? По умолчанию программа вызовет функцию `abort()`, однако такое поведение можно изменить. Мы вернемся к этой теме далее в главе.

Использование объектов в качестве исключений

Обычно функции, которые генерируют исключения, создают объекты. Преимущество такого подхода — возможность применения разных типов исключений для различения функций и ситуаций, генерирующих исключения. Кроме того, объект может содержать произвольную информацию, которая помогает определить причины, вызвавшие исключение. На основе этой информации блок `catch` может определить, какие действия следует предпринять. Ниже показан один из возможных вариантов исключения, генерируемого функцией `hmean()`:

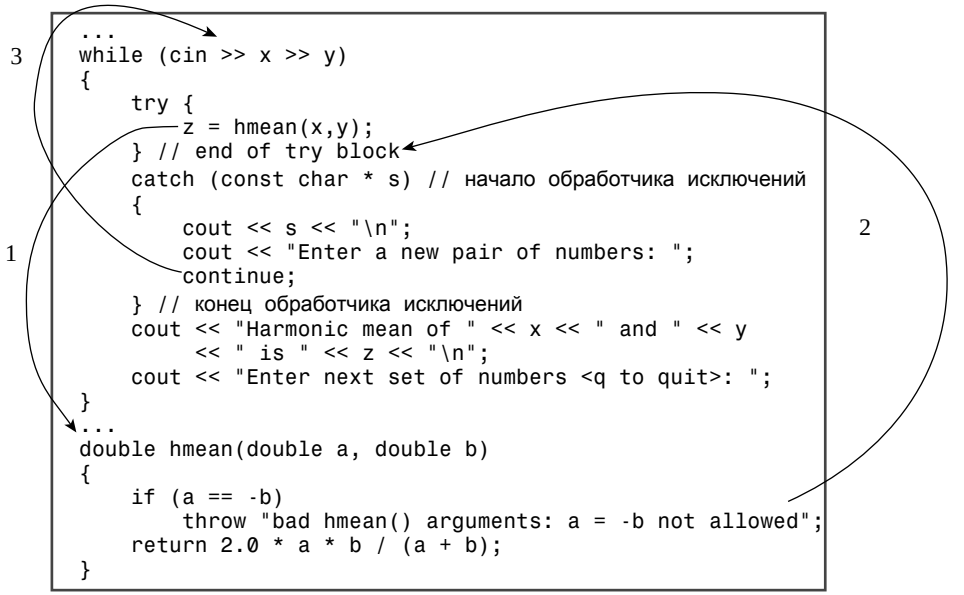
```
class bad_hmean
{
private:
    double v1;
    double v2;
public:
    bad_hmean(int a = 0, int b = 0) : v1(a), v2(b) {}
    void mesg();
};
```

```
inline void bad_hmean::mesg()
{
   std::cout << "hmean(" << v1 << ", " << v2 <<"): "
             << "invalid arguments: a = -b\n";  // недопустимые аргументы
}
```

Объект `bad_hmean` можно инициализировать значениями, переданными в `hmean()`, а для сообщения о возникшей проблеме удобен метод `mesg()`. В функции `hmean()` можно использовать примерно такой код:

```
if (a == -b)
   throw bad_hmean(a, b);
```

Здесь вызывается конструктор `bad_hmean()`, который инициализирует объект для сохранения значений аргументов.



1. Программа вызывает функцию `hmean()` внутри блока `try`.

2. `hmean()` генерирует исключение, передает управление в блок `catch` и присваивает значение строки исключения переменной `s`.

3. Блок `catch` возвращает управление в цикл `while`.

Рис. 15.2. Выполнение программы с исключениями

В листингах 15.10 и 15.11 добавлен еще один класс исключений `bad_gmean` и другая функция `gmean()`, которая генерирует исключение `bad_gmean`. Функция `gmean()` рассчитывает среднее геометрическое двух чисел — квадратный корень из их произведения. Эта функция определена, если оба аргумента неотрицательны. Поэтому она генерирует исключение, если обнаруживает отрицательные аргументы. Листинг 15.10 содержит заголовочный файл с определениями классов исключений, а листинг 15.11 — пример программы, использующей этот файл. Обратите внимание, что после блока `try` идут два блока `catch` подряд:

```
try {                       // начало блока try
   ...
}                           // конец  блока try
catch (bad_hmean & bg)      // начало блока catch
{
   ...
}
catch (bad_gmean & hg)
{
   ...
}                           // конец блока catch
```

Если hmean() сгенерирует исключение bad_hmean, его перехватит первый блок catch. Если же gmean() сгенерирует исключение bad_gmean, оно пройдет через первый блок catch и будет перехвачено вторым.

Листинг 15.10. exc_mean.cpp

```
// exc_mean.h -- классы исключений для hmean() и gmean()
#include <iostream>
class bad_hmean
{
private:
   double v1;
   double v2;
public:
   bad_hmean(double a = 0, double b = 0) : v1(a), v2(b){}
   void mesg();
};
inline void bad_hmean::mesg()
{
   std::cout << "hmean(" << v1 << ", " << v2 <<"): "
             << "invalid arguments: a = -b\n";   // неверные аргументы
}
class bad_gmean
{
public:
   double v1;
   double v2;
   bad_gmean(double a = 0, double b = 0) : v1(a), v2(b){}
   const char * mesg();
};
inline const char * bad_gmean::mesg()
{
   return "gmean() arguments should be >= 0\n"; // аргументы gmean() должны быть >= 0
}
```

Листинг 15.11. error4.cpp

```
// error4.cpp -- использование классов исключений
#include <iostream>
#include <cmath>  // или math.h, пользователям UNIX может потребоваться флаг -lm
#include "exc_mean.h"

// Прототипы функций
double hmean(double a, double b);
double gmean(double a, double b);
int main()
{
```

```
    using std::cout;
    using std::cin;
    using std::endl;
    double x, y, z;
    cout << "Enter two numbers: ";                  // запрос на ввод двух чисел
    while (cin >> x >> y)
    {
      try {                                         // начало блока try
        z = hmean(x,y);
        cout << "Harmonic mean of " << x << " and " << y
             << " is " << z << endl;                // вывод среднего гармонического
        cout << "Geometric mean of " << x << " and " << y
             << " is " << gmean(x,y) << endl;       // вывод среднего геометрического
        cout << "Enter next set of numbers <q to quit>: "; // ввод следующей пары чисел
      }                                             // конец блока try
      catch (bad_hmean & bg)                        // начало блока catch
      {
        bg.mesg();
        cout << "Try again.\n";                     // необходимо повторить попытку
        continue;
      }
      catch (bad_gmean & hg)
      {
        cout << hg.mesg();
        cout << "Values used: " << hg.v1 << ", "
             << hg.v2 << endl;                      // используемые знарчения
        cout << "Sorry, you don't get to play any more.\n";  // завершение работы
        break;
      }                                             // конец блока catch
    }
    cout << "Bye!\n";
    return 0;
}
double hmean(double a, double b)
{
    if (a == -b)
        throw bad_hmean(a,b);
    return 2.0 * a * b / (a + b);
}
double gmean(double a, double b)
{
    if (a < 0 || b < 0)
        throw bad_gmean(a,b);
    return std::sqrt(a * b);
}
```

Ниже приведен пример запуска программы из листингов 15.11 и 15.10; выполнение прекращается из-за недопустимых аргументов, переданных функции `gmean()`:

```
Enter two numbers: 4 12
Harmonic mean of 4 and 12 is 6
Geometric mean of 4 and 12 is 6.9282
Enter next set of numbers <q to quit>: 5 -5
hmean(5, -5): invalid arguments: a = -b
Try again.
5 -2
Harmonic mean of 5 and -2 is -6.66667
gmean() arguments should be >= 0
```

```
Values used: 5, -2
Sorry, you don't get to play any more.
Bye!
```

Обратите внимание, что обработчик bad_hmean использует оператор continue, а обработчик bad_gmean — оператор break. Поэтому при обнаружении недопустимых данных в функции hmean() программа пропускает остаток цикла и переходит к его началу, а недопустимые данные в функции gmean() приводят к прекращению цикла. Подобным образом программа определяет, какое исключение возникло (по типу исключения), и выбирает реакцию на исключение.

И еще одно: приемы, используемые в bad_gmean и bad_hmean, различны. В частности, в bad_gmean используются открытые данные и метод, возвращающий строку в стиле C.

Спецификации исключений в C++11

Иногда в целом хорошая идея не очень удачно реализуется на практике. Так случилось со *спецификациями исключений*, которые были добавлены в C++98, но в C++11 объявлены устаревшими. Это значит, что данное средство включено в стандарт, однако может быть изъято из него в будущем, поэтому лучше его не использовать.

Но прежде чем отказаться от спецификаций исключений, следует хотя бы узнать, что это такое. Выглядят они так:

```
double harm(double a) throw(bad_thing);      // может сгенерировать
                                             // исключение bad_thing
double marm(double) throw();                 // не генерирует исключения
```

Часть throw() — со списком типов или без него — является спецификацией исключений, и она должна присутствовать как в прототипе, так и в определении функции.

Одна из причин появления спецификаций исключений — необходимость уведомления пользователя о том, что может потребоваться блок try. Но это можно сделать и с помощью обычного комментария. Другая причина — разрешение компилятору на добавление кода, которые во время выполнения будет проверять, нарушена ли спецификация исключений. Это может случиться легко. Например, сама функция marm() может не генерировать исключения, но вызывать функцию, которая вызывает еще одну функцию, а та генерирует исключение. Возможно даже, что она ничего не генерировала на момент написания кода, но потом библиотека была обновлена, и теперь возможно появление исключений. Но все же в сообществе программистов — особенно среди тех, кто старательно пишет код, безопасный к исключениям — сложилось мнение, что эту возможность лучше игнорировать. И теперь и вы, с благословления C++11, можете также игнорировать ее.

Правда, в C++11 разрешена одна специальная спецификация — новое ключевое слова noexcept, которое указывает, что функция не генерирует исключений:

```
double marm() noexcept;  // marm() не генерирует исключений
```

О необходимости и пользе этой спецификации до сих пор идут споры, с некоторым уклоном в сторону того, что лучше избегать и ее (по крайней мере, в большинстве случаев). Но многие не боятся введения нового ключевого слова. Они считают, что знание о том, что функция не будет генерировать исключения, поможет компилятору оптимизировать код. Такую конструкцию можно рассматривать как обещание программиста о приличном поведении функции.

Существует также операция noexcept() (см. приложение Д), которая сообщает, может ли ее аргумент генерировать исключения.

Раскручивание стека

Предположим, что блок `try` не содержит непосредственного вызова функции, генерирующей исключение, однако он вызывает функцию, которая, в свою очередь, обращается к функции, генерирующей исключение. При этом управление передается из функции, в которой возникает исключение, в функцию, содержащую блок `try` и обработчики. Это называется *раскручиванием стека.*

Для начала посмотрим, как C++ обрабатывает вызовы функций и возвраты из них в нормальных обстоятельствах. Обычно при вызове функции информация о ней заносится в стек (см. главу 9).

В частности, в стеке сохраняется адрес инструкции вызывающей функции (*адрес возврата*). Когда вызываемая функция завершает свою работу, программа использует этот адрес для определения точки, с которой нужно продолжить выполнение программы. Аргументы функции также сохраняются в стеке и трактуются как автоматические переменные. Если вызванная функция создает новые автоматические переменные, то они тоже сохраняются в стеке. Если вызванная функция вызывает другую функцию, то ее информация также добавляется в стек и т.д. Когда выполнение функции завершается, управление передается по адресу, сохраненному в момент вызова этой функции, а вершина стека освобождается. То есть обычно функция возвращает управление в функцию, которая ее вызвала, при этом каждая функция при завершении освобождает свои автоматические переменные. Если автоматической переменной является объект класса, то вызывается соответствующий деструктор класса.

Предположим, что функция вместо нормального завершения прерывает свою работу с генерацией исключения. При этом программа, как положено, очищает стек. Но вместо перехода на ближайший адрес возврата в стеке программа продолжает очищать стек, пока не достигнет адреса возврата, который находится в блоке `try` (рис. 15.3). Затем управление передается в обработчики исключений за этим блоком, а не оператору, следующему за вызовом функции. Это процесс и называется *раскручиванием стека.*

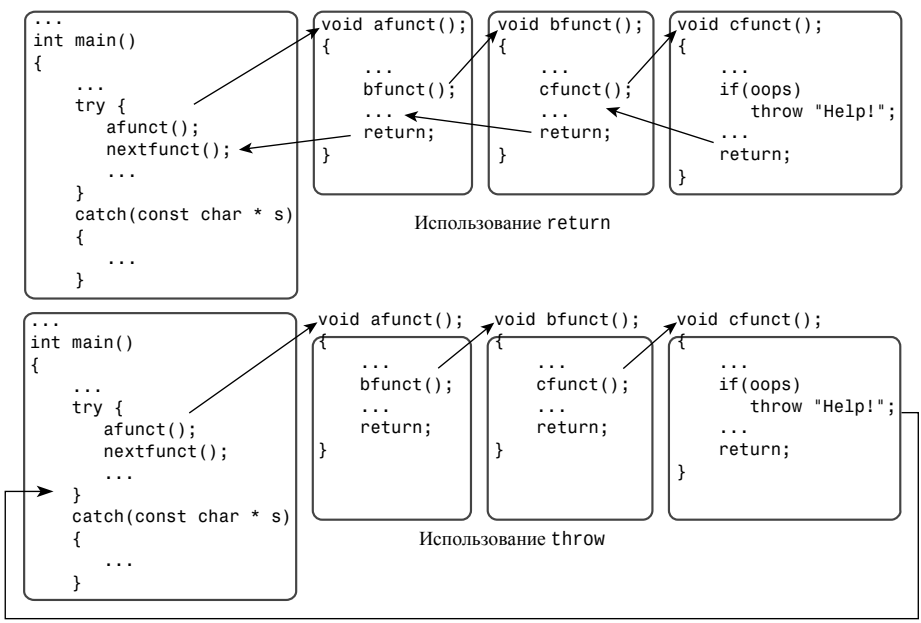


***Рис. 15.3.** Операторы `throw` и `return`*

Очень важной особенностью механизма операции throw является то, что, как и при возврате из функции, для всех автоматических объектов, которые находятся в стеке, вызываются деструкторы. Только при возврате из функции обрабатываются объекты, помещенные в стек лишь этой функцией, а оператор throw обрабатывает объекты, помещенные в стек целой цепочкой вызовов функций между блоком try и этим throw. Без раскручивания стека оператор throw не вызвал бы деструкторы для автоматических объектов, помещенных в стек промежуточными вызовами функций.

Пример раскручивания стека приведен в листинге 15.12. В нем функция main() вызывает функцию means(), которая, в свою очередь, вызывает hmean() и gmean(). Функция means() за неимением ничего лучшего вычисляет среднее из значений арифметического, гармонического и геометрического средних. Обе функции, main() и means(), создают объекты типа demo ("разговорчивый" класс, сообщающий, когда используются его конструктор и деструктор), так что можно видеть, что происходит с этими объектами при генерации исключений. Блок try в функции main() перехватывает оба исключения bad_hmean и bad_gmean, а блок try в функции means() перехватывает только исключение bad_hmean. Соответствующий код catch выглядит следующим образом:

```
catch (bad_hmean & bg)     // начало блока catch
{
   bg.mesg();
   std::cout << "Caught in means()\n";
   throw;                  // повторная генерация исключения
}
```

После реагирования на исключение и вывода сообщений код снова генерирует исключение, что в данном случае означает передачу исключения наверх, в функцию main(). (Обычно повторная генерация исключения приводит программу в следующую комбинацию try-catch, которая перехватывает данный тип исключения. При отсутствии обработчика исключения программа прекращает свое выполнение.) В листинге 15.12 используется тот же заголовочный файл, что и в листинге 15.11 — файл exc_mean.h из листинга 15.10.

Листинг 15.12. error5.cpp

```
// error5.cpp -- раскручивание стека
#include <iostream>
#include <cmath> // или math.h, пользователям UNIX может потребоваться флаг -lm
#include <string>
#include "exc_mean.h"
class demo
{
private:
   std::string word;
public:
   demo (const std::string & str)
   {
      word = str;
      std::cout << "demo " << word << " created\n";     // строка создана
   }
   ~demo()
   {
      std::cout << "demo " << word << " destroyed\n";   // строка уничтожена
   }
   void show() const
   {
      std::cout << "demo " << word << " lives!\n";      // строка существует
   }
};
```

```
// Прототипы функций
double hmean(double a, double b);
double gmean(double a, double b);
double means(double a, double b);

int main()
{
   using std::cout;
   using std::cin;
   using std::endl;

   double x, y, z;
   {
      demo d1("found in block in main()");
      cout << "Enter two numbers: ";          // запрос на ввод двух чисел
      while (cin >> x >> y)
      {
         try {                                // начало блока try
            z = means(x,y);
            cout << "The mean mean of " << x << " and " << y
                 << " is " << z << endl;      // вывод среднего из средних
            cout << "Enter next pair: ";      // ввод следующей пары
         }                                    // конец блока try
         catch (bad_hmean & bg)               // начало блока catch
         {
            bg.mesg();
            cout << "Try again.\n";          // необходимо повторить попытку
            continue;
         }
         catch (bad_gmean & hg)
         {
            cout << hg.mesg();
            cout << "Values used: " << hg.v1 << ", "
                 << hg.v2 << endl;
            cout << "Sorry, you don't get to play any more.\n";
            break;
         }                                    // конец блока catch
      }
      d1.show();
   }
   cout << "Bye!\n";
   cin.get();
   cin.get();
   return 0;
}

double hmean(double a, double b)
{
   if (a == -b)
      throw bad_hmean(a,b);
   return 2.0 * a * b / (a + b);
}

double gmean(double a, double b)
{
   if (a < 0 || b < 0)
      throw bad_gmean(a,b);
   return std::sqrt(a * b);
}
```

```
double means(double a, double b)
{
   double am, hm, gm;
   demo d2("found in means()");
   am = (a + b) / 2.0;          // среднее арифметическое
   try
   {
      hm = hmean(a,b);
      gm = gmean(a,b);
   }
   catch (bad_hmean & bg)       // начало блока catch
   {
      bg.mesg();
      std::cout << "Caught in means()\n";
      throw;                    // повторная генерация исключения
   }
   d2.show();
   return (am + hm + gm) / 3.0;
}
```

Ниже показан пример запуска программы из листингов 15.12 и 15.10:

```
demo found in block in main() created
Enter two numbers: 6 12
demo found in means() created
demo found in means() lives!
demo found in means() destroyed
The mean mean of 6 and 12 is 8.49509
6 -6
demo found in means() created
hmean(6, -6): invalid arguments: a = -b
Caught in means()
demo found in means() destroyed
hmean(6, -6): invalid arguments: a = -b
Try again.
6 -8
demo found in means() created
demo found in means() destroyed
gmean() arguments should be >= 0
Values used: 6, -8
Sorry, you don't get to play any more.
demo found in block in main() lives!
demo found in block in main() destroyed
Bye!
```

Замечания по программе

Давайте проанализируем выполнение программы из предыдущего раздела. Сначала в `main()` с помощью конструктора `demo` создается объект. Потом вызывается функция `means()` и создается другой объект `demo`. Функция `means()` передает значения `6` и `12` в `hmean()` и `gmean()`, а эти функции возвращают значения, на основе которых рассчитывается и возвращается результат. Прежде чем вернуть результат, `means()` вызывает `d2.show()`. Вернув результат, `means()` завершается и автоматически вызывается деструктор для `d2`:

```
demo found in means() lives!
demo found in means() destroyed
```

Следующий цикл ввода передает в means() значения 6 и -6, после чего функция means() создает новый объект demo и передает значения в hmean().

Функция hmean() генерирует исключение bad_hmean, которое в means() перехватывается блоком catch, что видно в следующем фрагменте:

```
hmean(6, -6): invalid arguments: a = -b
Caught in means()
```

Оператор throw в этом блоке завершает выполнение means() и передает исключение в main(). Вызов d2.show не выполняется — т.е. means() завершена. Однако деструктор для d2 вызывается:

```
demo found in means() destroyed
```

Это демонстрирует очень важный аспект исключений: когда программа раскручивает стек до места перехвата исключения, она очищает автоматические переменные класса, сохраненные в стеке. Если переменная является объектом класса, вызывается деструктор этого класса.

Повторно сгенерированное исключение добирается до main(), где захватывается и обрабатывается соответствующим блоком catch:

```
hmean(6, -6): invalid arguments: a = -b
Try again.
```

Третий цикл ввода передает в means() значения 6 и -8. И снова means() создаст новый объект demo. Он передает значения 6 и -8 в функцию hmean(), которая без проблем их обрабатывает. Затем hmean() передает 6 и -8 в функцию gmean(), которая генерирует исключение bad_gmean. Поскольку means() не перехватывает это исключение, она передает его дальше в main(), и никакой код из means() больше не выполнятся. Но и в этом случае при раскручивании стека освобождаются локальные автоматические переменные, в частности, вызывается деструктор для d2:

```
demo found in means() destroyed
```

Наконец, обработчик bad_gmean в main() перехватывает исключение и завершает цикл:

```
gmean() arguments should be >= 0
Values used: 6, -8
Sorry, you don't get to play any more.
```

Далее программа нормально завершается, выводит несколько сообщений и автоматически вызывает деструктор для d1. Если блок catch вызовет, скажем, exit(EXIT_FAILURE), а не break, программа завершится немедленно, и вы не увидите сообщений:

```
demo found in main() lives!
Bye!
```

Однако будет выдано сообщение:

```
demo found in main() destroyed
```

Здесь механизм исключений также освободит автоматические переменные, помещенные в стек.

Обратите внимание на спецификацию исключения для means():

```
double means(double a, double b) throw(bad_hmean, bad_gmean);
```

Дополнительные свойства исключений

Хотя механизм throw-catch имеет много общего с аргументами и механизмом возврата функций, существуют и отличия. Одно мы уже рассмотрели: оператор возврата из функции передает управление вызвавшей функции, а оператор throw передает управление по цепочке в первую функцию, содержащую комбинацию try-catch, которая перехватывает данное исключение. Например, когда в листинге 15.12 функция hmean() генерирует исключение, управление передается наверх в means(), но когда gmean() генерирует исключение, управление передается в main().

Другое отличие состоит в том, что когда компилятор генерирует исключение, он всегда создает временную копию, даже если спецификатор исключения и блок catch задают ссылку:

```
class problem {...};
...
void super() throw (problem)
{
   ...
   if (oh_no)
   {
      problem oops; // создание объекта исключения
      throw oops;   // генерация исключения
      ...
}
...
try {
   super();
}
catch(problem & p)
{
   // операторы
}
```

Здесь p ссылается на копию oops, а не на сам oops. Это хорошо, потому что после завершения метода super() исключение oops уже не существует. Кстати, удобнее объединить создание исключения с оператором throw:

```
throw problem();     // создание и генерация стандартного объекта problem
```

Вы можете спросить, зачем в коде используется ссылка, если throw генерирует копию. Ведь обычно ссылочные величины возвращаются, чтобы не создавать копию объекта. Однако ссылки обладают еще одним важным свойством: ссылка на базовый класс может ссылаться на объекты производных классов. Предположим, что существует коллекция типов исключений, которые связаны наследованием. В этом случае в спецификации исключения нужна только ссылка на базовый тип, которая будет соответствовать любому исключению, сгенерированному производными объектами.

Допустим, что имеется иерархия классов исключений, и разные типы исключений нужно обрабатывать по-разному. Ссылка на базовый класс может перехватывать все объекты семейства, но объект производного класса может перехватить только объект этого класса и классов, производных от него. Сгенерированный объект будет перехвачен первым же соответствующим блоком catch. Значит, блоки catch следует располагать в порядке, обратном порождению:

```
class bad_1 {...};
class bad_2 : public bad_1 {...};
class bad_3 : public bad_2 {...};
```

```
...
void duper() throw (bad_1)
{
   ...
   if (oh_no)
      throw bad_1();
   if (rats)
      throw bad_2();
   if (drat)
      throw bad_3();
}
...
try {
   duper();
}
catch(bad_3 &be)
{ // операторы }
catch(bad_2 &be)
{ // операторы}
catch(bad_1 &be)
{ // операторы }
```

Если обработчик `bad_1 &` будет первым, он перехватит исключения `bad_1`, `bad_2` и `bad_3`. При обратном порядке расположения обработчиков исключение `bad_3` будет перехвачено обработчиком `bad_3 &`.

Совет

Если имеется иерархия наследования классов исключений, необходимо расположить блоки `catch` в таком порядке, чтобы исключение самого последнего производного класса перехватывалось первым, а исключение базового класса — последним.

Расположение блоков `catch` в соответствующем порядке позволяет определить, как будут обрабатываться исключения различных типов. Но иногда невозможно предвидеть тип возможных исключений. Предположим, что создается функция, которая вызывает другую функцию, и неизвестно, генерирует ли эта функция исключение. Оказывается, можно перехватить исключение, даже не зная его тип. Хитрость заключается в использовании многоточия вместо типа исключения:

```
catch (...) { // операторы }  // перехватывает исключение любого типа
```

Если тип некоторых исключений известен, такую универсальную ловушку можно расположить в конце блока `catch` — вроде `default` в операторе `switch`:

```
try {
   duper();
}
catch(bad_3 &be)
{ // операторы }
catch(bad_2 &be)
{ // операторы }
catch(bad_1 &be)
{ // операторы }
catch(bad_hmean & h)
{ // операторы }
catch (...)      // перехват всего, что осталось
{ // операторы }
```

Вместо ссылок можно перехватывать непосредственно объекты исключений. Ловушка для базового класса будет перехватывать объекты производного класса, но свойства, характерные для производного объекта, при этом будут скрыты. Поэтому будут использоваться виртуальные методы базового класса.

Класс exception

Главная цель введения исключений в C++ — создать средства на уровне языка для разработки надежных программ. Ведь исключения упрощают обработку ошибок в программах, при этом не нужны другие, менее удобные, способы. Гибкость и относительное удобство исключений поощряют программистов к использованию этого механизма в своих разработках. В общем, исключения — это возможность, которая, как и классы, может изменить сам подход к программированию.

В новых компиляторах C++ исключения входят в состав языка. Например, в заголовочном файле exception (ранее exception.h или except.h) определен класс exception, который служит в C++ базовым классом для других классов исключений. В коде можно генерировать объект exception или применять класс exception в качестве базового класса. Среди виртуальных функций-членов имеется функция what(), возвращающая строку, природа которой зависит от реализации. Поскольку этот метод виртуальный, его можно переопределить в производном классе:

```
#include <exception>
class bad_hmean : public std::exception
{
public:
   const char * what() { return "bad arguments to hmean()"; }
   ...
};
class bad_gmean : public std::exception
{
public:
   const char * what() { return "bad arguments to gmean()"; }
   ...
};
```

Если нет необходимости в индивидуальной обработке этих производных исключений, их можно перехватить в обработчике базового класса:

```
try {
   ...
}
catch(std::exception & e)
{
   cout << e.what() << endl;
   ...
}
```

Или же можно перехватывать различные типы исключений по отдельности.

В библиотеке C++ определено много типов исключений, основанных на классе exception.

Классы исключений stdexcept

В заголовочном файле stdexcep определено еще несколько классов исключений. Первым делом, в нем определены классы logic_error и runtime_error, оба общедоступно порожденные от exception:

```
class logic_error : public exception {
public:
   explicit logic_error(const string& what_arg);
   ...
};
class domain_error : public logic_error {
public:
   explicit domain_error(const string& what_arg);
   ...
};
```

Обратите внимание, что конструкторы принимают в качестве аргумента объект `string`; этот аргумент содержит символьные данные в виде строки стиля C, которую возвращает метод `what()`.

Эти два новых класса служат основой для двух семейств производных классов. Семейство `logic_error`, как понятно из названия, описывает типичные логические ошибки. В принципе, при аккуратном программировании таких ошибок можно избежать, но на практике они все же возникают. По имени класса можно определить вид ошибок, для которых он предназначен:

```
domain_error
invalid_argument
length_error
out_of_bounds
```

У каждого класса имеется конструктор, как у `logic_error`, который позволяет указать строку, возвращаемую методом `what()`.

Возможно, здесь будет удобна математическая аналогия. Математическая функция имеет область определения и область значений. Область определения состоит из значений, для которых определена функция, а область значений — из значений, которые функция возвращает. Например, область определения функции синуса — от минус бесконечности до плюс бесконечности, поскольку синус определен для всех вещественных чисел. Но область значений функции синуса — от –1 до +1, поскольку это экстремальные значения синуса. Область определения обратной функции, арксинуса, является отрезком от –1 до +1, а область возвращаемых значений — от $-\pi$ до $+\pi$. Если написать функцию, которая передает аргумент в функцию `std::asin()`, то эта функция может сгенерировать объект `domain_error`, если аргумент окажется вне области определения от –1 до +1.

Исключение `invalid_argument` сообщает, что функции было передано непредвиденное значение. Например, если функция ожидает получить строку, состоящую только из символов `'1'` или `'0'`, она может сгенерировать исключение `invalid_argument`, если обнаружит в строке другой символ.

Исключение `length_error` используется, если для какого-то действия недостаточно памяти. Например, класс `string` содержит метод `append()`, который генерирует исключение `length_error`, если результирующая строка получится длиннее максимально допустимой величины.

Исключение `out_of_bounds` обычно служит для обозначения ошибок индексации. Например, можно определить класс, подобный массиву, для которого `operator()[]` сгенерирует исключение `out_of_bounds` в том случае, если применяемый индекс является недопустимым для этого массива.

Семейство `runtime_error` предназначено для ошибок, которые могут возникнуть во время выполнения программы, но не могут быть предсказаны и выявлены заранее. Имя каждого класса определяет вид ошибок, для которых он предназначен:

```
range_error
overflow_error
underflow_error
```

У каждого класса имеется конструктор, похожий на конструктор runtime_error, который позволяет задать строку, возвращаемую методом what().

Ошибка потери значимости (underflow_error) может возникнуть в вычислениях с плавающей точкой. Существует наименьшая ненулевая положительная величина, которая может быть представлена типом с плавающей точкой. Вычисления, при которых возникают меньшие значения, приведут к генерации исключения потери значимости. Ошибка переполнения (overflow_error) возникает для целых типов или типов с плавающей точкой, если абсолютная величина результата превышает максимально возможное значение для этих типов. Результат вычисления может лежать вне допустимого диапазона без потери значимости или переполнения, в этом случае можно использовать исключение range_error.

В общем случае исключение из семейства logic_error отражает проблему, которую предположительно можно устранить в коде программы, а исключение из семейства runtime_error — ошибку, избежать которой нельзя. Оба эти класса ошибок обладают сходными характеристиками. Основное различие в том, что разные имена классов позволяют отдельно обрабатывать разные типы исключений. С другой стороны, отношения наследования позволяют при желании объединить эти классы воедино. Например, следующий код перехватывает исключение out_of_bounds отдельно, обрабатывает остальное семейство исключений logic_error как группу, а семейство объектов runtime_error и все оставшиеся объекты, производные от exception, обрабатывает коллективно:

```
try {
   ...
}
catch(out_of_bounds & oe) // перехват ошибки out_of_bounds
{...}
catch(logic_error & oe)   // перехват остальных ошибок семейства logic_error
{...}
catch(exception & oe)     // перехват runtime_error и других объектов exception
{...}
```

Если какой-либо из этих библиотечных классов не соответствует вашим требованиям, от logic_error или runtime_error можно породить новый класс исключения, который войдет в общую иерархию.

Исключение bad_alloc и операция new

В настоящее время в C++ проблемы, возникающие во время выделения памяти с помощью операции new, обрабатываются путем генерации в new исключения bad_alloc. Заголовочный файл new включает объявление класса bad_alloc, открыто унаследованного от класса exception. Правда, в давние времена операция new возвращала нулевой указатель, если не могла выделить запрошенный объем памяти.

Текущий подход демонстрируется в листинге 15.13. Если исключение перехвачено, программа выводит зависящее от реализации сообщение, которое возвращается унаследованным методом what(), и затем завершается.

Листинг 15.13. newexcp.cpp

```
// newexcp.cpp -- исключение bad_alloc
#include <iostream>
```

```
#include <new>
#include <cstdlib>  // для exit(), EXIT_FAILURE
using namespace std;
struct Big
{
   double stuff[20000];
};
int main()
{
   Big * pb;
   try {
      cout << "Trying to get a big block of memory:\n";
           // Попытка выделения крупного блока памяти
      pb = new Big[10000];                          // 1 600 000 000 байт
      cout << "Got past the new request:\n";        // вывод результатов запроса new
   }
   catch (bad_alloc & ba)
   {
      cout << "Caught the exception!\n";            // произошло исключение
      cout << ba.what() << endl;
      exit(EXIT_FAILURE);
   }
   cout << "Memory successfully allocated\n";      // память успешно выделена
   pb[0].stuff[0] = 4;
   cout << pb[0].stuff[0] << endl;
   delete [] pb;
   return 0;
}
```

Ниже показан вывод этой программы в одной из систем:

```
Trying to get a big block of memory:
Caught the exception!
std::bad_alloc
```

В этом случае метод `what()` возвращает строку `"std::bad_alloc"`.

Если программа выполнилась без ошибок, можно попробовать увеличить объем запрашиваемой памяти.

Нулевой указатель и операция `new`

К этому моменту уже написан большой объем кода, когда (старая) операция `new` возвращала в случае сбоя нулевой указатель. В некоторых компиляторах предусмотрен флаг, который позволяет пользователям выбрать удобное для них поведение операции `new`. В текущем стандарте языка имеется альтернативный вариант `new`, который по-прежнему возвращает нулевой указатель. Он используется примерно так:

```
int * pi = new (std::nothrow) int;
int * pa = new (std::nowthrow) int[500];
```

Эта форма позволяет переписать основную часть листинга 15.13 следующим образом:

```
Big * pb;
pb = new (std::nothrow) Big[10000];  // 1 600 000 000 байт
if (pb == 0)
{
   cout << "Could not allocate memory. Bye.\n";
   exit(EXIT_FAILURE);
}
```

Исключения, классы и наследование

Исключения, классы и наследование взаимодействуют несколькими способами. Во-первых, можно породить один класс исключения от другого класса, как это сделано в стандартной библиотеке C++. Во-вторых, можно добавить исключения в классы, вставив объявление класса исключения в определение класса. В-третьих, такое вложенное объявление может быть унаследовано и само служить базовым классом.

Код в листинге 15.14 предназначен для исследования таких возможностей. В этом заголовочном файле объявлен простой класс `Sales`, содержащий значение года, и массив из 12 ежемесячных объемов продаж. Класс `LabeledSales` порожден от класса `Sales` и содержит дополнительный член для хранения метки данных.

Листинг 15.14. `sales.h`

```
// sales.h -- исключения и наследование
#include <stdexcept>
#include <string>
class Sales
{
public:
   enum {MONTHS = 12};       // может быть статической константой
   class bad_index : public std::logic_error
   {
   private:
      int bi;                // недопустимое значение индекса
   public:
      explicit bad_index(int ix,
      const std::string & s = "Index error in Sales object\n");
                             // Ошибка индекса в объекте Sales
      int bi_val() const {return bi;}
      virtual ~bad_index() throw() {}
   };
   explicit Sales(int yy = 0);
   Sales(int yy, const double * gr, int n);
   virtual ~Sales() { }
   int Year() const { return year; }
   virtual double operator[](int i) const;
   virtual double & operator[](int i);
private:
   double gross[MONTHS];
   int year;
};
class LabeledSales : public Sales
{
public:
   class nbad_index : public Sales::bad_index
   {
   private:
      std::string lbl;
   public:
      nbad_index(const std::string & lb, int ix,
                 const std::string & s = "Index error in LabeledSales object\n");
      const std::string & label_val() const {return lbl;}
      virtual ~nbad_index() throw() {}
   };
   explicit LabeledSales(const std::string & lb = "none", int yy = 0);
   LabeledSales(const std::string & lb, int yy, const double * gr, int n);
```

```
    virtual ~LabeledSales() { }
    const std::string & Label() const {return label;}
    virtual double operator[](int i) const;
    virtual double & operator[](int i);
private:
    std::string label;
};
```

Рассмотрим некоторые особенности кода в листинге 15.14. Символьная константа MONTHS расположена в защищенном разделе Sales, поэтому ее значение доступно для производных классов, таких как LabeledSales.

Класс bad_index находится в открытом разделе Sales; это делает его доступным в качестве типа для клиентских блоков catch. Правда, вовне этот тип нужно указывать как Sales::bad_index. Данный класс порожден от стандартного класса logic_error, он может сохранять недопустимые значения индексов и сообщать о них.

Класс nbad_index находится в открытом разделе LabeledSales и доступен в клиентском коде как LabeledSales::nbad_index.

Он порожден от bad_index и может дополнительно сохранять и выводить метки объектов LabeledSales. Поскольку класс bad_index порожден от logic_error, то и nbad_index также порожден от logic_error.

Оба класса содержат перегруженные методы operator[](), которые предназначены для доступа к хранимым в объекте отдельным элементам массива и для генерации исключения, если индекс массива выходит за допустимые пределы.

Оба класса, bad_index и nbad_index, используют спецификацию исключения throw(). Причина в том, что оба они, в конечном счете, унаследованы от базового класса exception, виртуальный деструктор которого использует спецификацию исключения throw(). Это характерно для C++98; в C++11 деструктор exception не имеет спецификации исключения.

В листинге 15.15 показана реализация методов, которые не были встроенным образом определены в листинге 15.14. Обратите внимание, что вложенные классы требуют многократного использования операции разрешения контекста. Учтите также, что функции operator[]() генерируют исключения при выходе индекса за пределы массива.

Листинг 15.15. sales.cpp

```
// sales.cpp -- реализация Sales
#include "sales.h"
using std::string;
Sales::bad_index::bad_index(int ix, const string & s )
    : std::logic_error(s), bi(ix)
{
}
Sales::Sales(int yy)
{
    year = yy;
    for (int i = 0; i < MONTHS; ++i)
        gross[i] = 0;
}
Sales::Sales(int yy, const double * gr, int n)
{
    year = yy;
    int lim = (n < MONTHS)? n : MONTHS;
```

```
    int i;
    for (i = 0; i < lim; ++i)
        gross[i] = gr[i];
    // Для i > n и i < MONTHS
    for ( ; i < MONTHS; ++i)
        gross[i] = 0;
}
double Sales::operator[](int i) const
{
    if(i < 0 || i >= MONTHS)
        throw bad_index(i);
    return gross[i];
}
double & Sales::operator[](int i)
{
    if(i < 0 || i >= MONTHS)
        throw bad_index(i);
    return gross[i];
}
LabeledSales::nbad_index::nbad_index(const string & lb, int ix,
                                const string & s ) : Sales::bad_index(ix, s)
{
    lbl = lb;
}
LabeledSales::LabeledSales(const string & lb, int yy)
          : Sales(yy)
{
    label = lb;
}
LabeledSales::LabeledSales(const string & lb, int yy,
                        const double * gr, int n)
          : Sales(yy, gr, n)
{
    label = lb;
}
double LabeledSales::operator[](int i) const
{
    if(i < 0 || i >= MONTHS)
        throw nbad_index(Label(), i);
    return Sales::operator[](i);
}
double & LabeledSales::operator[](int i)
{
    if(i < 0 || i >= MONTHS)
        throw nbad_index(Label(), i);
    return Sales::operator[](i);
}
```

Эти классы используются в программе из листинга 15.16, которая сначала пытается выйти за пределы массива в объекте `LabeledSales` по имени `sales2`, а затем — за пределы массива в объекте `Sales` по имени `sales1`. Эти попытки реализованы в двух разных блоках `try`, которые проверяют каждый вид исключений.

Листинг 15.16. `use_sales.cpp`

```
// use_sales.cpp -- вложенные исключения
#include <iostream>
#include "sales.h"
```

```
int main()
{
   using std::cout;
   using std::cin;
   using std::endl;
   double vals1[12] =
   {
      1220, 1100, 1122, 2212, 1232, 2334,
      2884, 2393, 3302, 2922, 3002, 3544
   };
   double vals2[12] =
   {
      12, 11, 22, 21, 32, 34,
      28, 29, 33, 29, 32, 35
   };
   Sales sales1(2011, vals1, 12);
   LabeledSales sales2("Blogstar",2012, vals2, 12 );
   cout << "First try block:\n";                          // первый блок try
   try
   {
      int i;
      cout << "Year = " << sales1.Year() << endl;        // год
      for (i = 0; i < 12; ++i)
      {
         cout << sales1[i] << ' ';
         if (i % 6 == 5)
            cout << endl;
      }
      cout << "Year = " << sales2.Year() << endl;        // год
      cout << "Label = " << sales2.Label() << endl;      // метка
      for (i = 0; i <= 12; ++i)
      {
         cout << sales2[i] << ' ';
         if (i % 6 == 5)
            cout << endl;
      }
      cout << "End of try block 1.\n";                   // конец первого блока try
   }
   catch(LabeledSales::nbad_index & bad)
   {
      cout << bad.what();
      cout << "Company: " << bad.label_val() << endl;    // компания
      cout << "bad index: " << bad.bi_val() << endl;     // недопустимый индекс
   }
   catch(Sales::bad_index & bad)
   {
      cout << bad.what();
      cout << "bad index: " << bad.bi_val() << endl;     // недопустимый индекс
   }
   cout << "\nNext try block:\n";                        // второй блок try
   try
   {
      sales2[2] = 37.5;
      sales1[20] = 23345;
      cout << "End of try block 2.\n";                   // конец второго блока try
   }
   catch(LabeledSales::nbad_index & bad)
   {
```

```
        cout << bad.what();
        cout << "Company: " << bad.label_val() << endl;    // компания
        cout << "bad index: " << bad.bi_val() << endl;     // недопустимый индекс
    }
    catch(Sales::bad_index & bad)
    {
        cout << bad.what();
        cout << "bad index: " << bad.bi_val() << endl;     // недопустимый индекс
    }
    cout << "done\n";
    return 0;
}
```

Ниже показан вывод программы из листингов 15.14, 15.15 и 15.16:

```
First try block:
Year = 2011
1220 1100 1122 2212 1232 2334
2884 2393 3302 2922 3002 3544
Year = 2012
Label = Blogstar
12 11 22 21 32 34
28 29 33 29 32 35
Index error in LabeledSales object
Company: Blogstar
bad index: 12

Next try block:
Index error in Sales object
bad index: 20
done
```

Потеря исключений

После того как исключение сгенерировано, у него есть две возможности вызвать проблемы. Если исключение сгенерировано в функции, имеющей спецификацию исключения, оно должно соответствовать одному из типов в списке спецификации. (Вспомните, что в иерархии наследования тип класса соответствует объектам этого класса и производных от него классов.) Если исключение не соответствует спецификации, оно называется *непредвиденным исключением* и по умолчанию приводит к останову программы. (Хотя в C++11 спецификации исключений объявлены устаревшими, они по-прежнему остались в языке и кое-где в существующем коде.) Если исключение преодолевает этот первый барьер (или избегает его, в силу отсутствия спецификации исключения), то оно должно быть перехвачено. Если исключение не перехвачено, что может произойти при отсутствии блока `try` или соответствующего блока `catch`, оно называется *неперехваченным исключением*. По умолчанию такое исключение приводит к останову программы. Однако можно изменить реакцию программы на непредвиденные и неперехваченные исключения. Посмотрим, как это делается, и начнем с неперехваченного исключения.

Неперехваченное исключение не приводит к немедленному останову программы. Вместо этого программа сначала обращается к функции по имени `terminate()`. По умолчанию функция `terminate()` вызывает функцию `abort()`. Но можно изменить поведение функции `terminate()`, *зарегистрировав* функцию, которую `terminate()` будет вызывать вместо `abort()`. Для этого необходимо вызвать функцию

set_terminate(). Обе функции, terminate() и set_terminate(), объявлены в заголовочном файле exception:

```
typedef void (*terminate_handler)();
terminate_handler set_terminate(terminate_handler f) throw();   // C++98
terminate_handler set_terminate(terminate_handler f) noexcept;  // C++11
void terminate();                                                // C++98
void terminate() noexcept;                                       // C++11
```

Здесь оператор typedef объявляет terminate_handler именем типа указателя на функцию, которая не принимает аргументы и не возвращает значение. Функция set_terminate() принимает в качестве аргумента имя функции (т.е. ее адрес) без аргументов и возвращает тип void. После вызова set_terminate() возвращает адрес функции, зарегистрированной перед этим. Если вызвать функцию set_terminate() более одного раза, terminate() вызовет функцию, зарегистрированную самым последним вызовом set_terminate().

Рассмотрим пример. Предположим, что потребовалось неперехватываемое исключение, которое выводит сообщение об этом, затем вызывает функцию exit() с состоянием завершения в 5. Сначала потребуется включить в проект заголовочный файл exception. Объявления этого файла можно сделать доступными с помощью директивы using, подходящих объявлений using или просто квалификатора std::.

```
#include <exception>
using namespace std;
```

После этого нужно создать функцию, которая выполняет два требуемых действия и имеет подходящий прототип:

```
void myQuit()
{
   cout << "Terminating due to uncaught exception\n";
   exit(5);
}
```

И, наконец, в начале программы необходимо указать эту функцию как выбранное действие завершения программы:

```
set_terminate(myQuit);
```

В результате, если исключение будет сгенерировано и не перехвачено, программа вызовет terminate(), а terminate() вызовет MyQuit().

Теперь рассмотрим непредвиденные исключения. Спецификации исключений в функции дают возможность пользователям функции узнать, какие исключения перехватывать. Предположим, что имеется следующий прототип:

```
double Argh(double, double) throw(out_of_bounds);
```

Тогда функцию можно использовать следующим образом:

```
try {
   x = Argh(a, b);
}
catch(out_of_bounds & ex)
{
   ...
}
```

Хорошо, когда известно, какие исключения перехватывать; вспомните, что неперехваченное исключение по умолчанию приводит к останову программы.

Но это еще не все. В принципе, спецификация исключений должна содержать исключения, генерируемые функциями, которые вызываются рассматриваемой функцией. Например, если `Argh()` вызывает функцию `Duh()`, которая генерирует объект исключения `retort`, то `retort` должен быть указан в спецификации исключений обеих функций `Argh()` и `Duh()`. Если вы не сами (или не тщательно) реализуете все функции, то нет гарантии, что все будет работать правильно. Например, может возникнуть необходимость в старых коммерческих библиотеках, которые не содержат спецификации исключений. Это означает, что нужно очень хорошо продумать, что получится, если функция сгенерирует исключение, которое не указано в спецификации. (Это также означает, что механизм спецификаций исключений в целом может оказаться весьма громоздким, из-за чего он и не рекомендуется к применению в C++11.)

Такое поведение сильно похоже на неперехваченное исключение. При возникновении непредвиденного исключения программа вызывает функцию `unexpected()`. Эта функция, в свою очередь, обращается к функции `terminate()`, которая по умолчанию вызывает `abort()`. По аналогии с функцией `set_terminate()`, изменяющей поведение `terminate()`, имеется и функция `set_unexpected()`, которая модифицирует поведение `unexpected()`. Эти новые функции также объявлены в заголовочном файле `exception`:

```
typedef void (*unexpected_handler)();
unexpected_handler set_unexpected(unexpected_handler f) throw();    // C++98
unexpected_handler set_unexpected(unexpected_handler f) noexcept;   // C++11
void unexpected();                                                  // C++98
void unexpected() noexcept;                                         // C++11
```

Однако поведение функции, которая указывается в `set_unexpected()`, более управляемо. В частности, функция `unexpected_handler` может:

- завершить программу, вызвав `terminate()` (по умолчанию), `abort()` или `exit()`;
- сгенерировать исключение.

Результат генерации исключения (второй вариант) зависит от исключения, сгенерированного функцией замены `unexpected_handler`, и исходной спецификации исключений для функции, которая сгенерировала непредвиденный тип исключения.

- Если вновь сгенерированное исключение соответствует исходной спецификации исключений, то программа выполняется нормально — ищет блок `catch`, который соответствует возникшему исключению. По сути, при таком подходе непредвиденный тип исключения меняется на ожидаемый тип.
- Если вновь сгенерированное исключение не соответствует исходной спецификации исключений и если спецификация исключений *не содержит* тип `std::bad_exception`, то программа вызывает `terminate()`. Тип `bad_exception` порожден от типа `exception`, а его объявление содержится в заголовочном файле `exception`.
- Если вновь сгенерированное исключение не соответствует исходной спецификации исключений и если спецификация исключений *содержит* тип `std::bad_exception`, то непредвиденное исключение заменяется исключением типа `std::bad_exception`.

В общем, если необходимо перехватывать все исключения — ожидаемые и непредвиденные — можно поступить примерно так. Первым делом, нужно сделать доступными объявления заголовочного файла исключений:

```
#include <exception>
using namespace std;
```

Затем следует создать функцию замены, которая преобразует непредвиденные исключения в тип bad_exception с соответствующим прототипом:

```
void myUnexpected()
{
   throw std::bad_exception();  // или просто throw;
}
```

Применение throw без указания исключения приведет к повторной генерации первоначального исключения. Но если спецификация исключений содержит этот тип, исключение будет заменено объектом bad_exception.

Далее в начале программы нужно указать функцию в качестве выбранного действия в ответ на непредвиденное исключение:

```
set_unexpected(myUnexpeced);
```

И, наконец, необходимо включить тип bad_exception в спецификацию исключений и цепочку блоков catch:

```
double Argh(double, double) throw(out_of_bounds, bad_exception);
...
try {
   x = Argh(a, b);
}
catch(out_of_bounds & ex)
{
   ...
}
catch(bad_exception & ex)
{
   ...
}
```

Предостережения относительно использования исключений

Из предыдущего обсуждения можно сделать (в общем-то, справедливый) вывод, что обработка исключений должна быть встроена в программу, а не присоединена извне. Однако у такого подхода есть свои недостатки. Например, исключения увеличивают размер программы и снижают скорость ее выполнения. Спецификации исключений плохо сочетаются с шаблонами, т.к. шаблонные функции могут генерировать различные виды исключений, в зависимости от конкретной специализации. Динамическое распределение памяти также не всегда гладко взаимодействует с исключениями.

Рассмотрим немного подробнее совместную работу динамического распределения памяти и исключений. Пусть имеется следующая функция:

```
void test1(int n)
{
   string mesg("I'm trapped in an endless loop");
   ...
   if (oh_no)
      throw exception();
   ...
   return;
}
```

Класс string использует динамическое распределение памяти. Обычно деструктор string вызывается, когда функция доходит до оператора return и завершается. Благодаря раскручиванию стека оператор throw позволяет вызвать деструктор, хотя

он и завершает функцию преждевременно. То есть в этом случае управление памятью выполняется без проблем.

Теперь рассмотрим следующую функцию:

```
void test2(int n)
{
   double * ar = new double[n];
   ...
   if (oh_no)
      throw exception();
   ...
   delete [] ar;
   return;
}
```

А вот здесь проблема присутствует. Раскручивание стека удаляет переменную `ar` из стека. Однако из-за преждевременного завершения функции оператор `delete []` в конце тела функции будет пропущен. То есть указатель уничтожен, а память, на которую он указывал, остается выделенной, хотя обратиться к ней невозможно. Короче говоря, имеется утечка памяти.

Такую утечку можно устранить. Например, можно перехватить исключение в той же функции, которая его сгенерировала, добавить в блок `catch` код очистки и сгенерировать исключение заново:

```
void test3(int n)
{
   double * ar = new double[n];
   ...
   try {
      if (oh_no)
         throw exception();
   }
   catch(exception & ex)
   {
      delete [] ar;
      throw;
   }
   ...
   delete [] ar;
   return;
}
```

Однако такой подход при недостаточной внимательности может породить новые ошибки. Другой способ основан на использовании шаблонов интеллектуальных указателей, которые рассматриваются в главе 16.

В общем, несмотря на исключительную важность управления исключениями в некоторых проектах, оно требует дополнительных усилий по программированию, увеличивает размер программы и замедляет ее работу. Правда, ущерб от отсутствия таких проверок может быть гораздо бо́льшим.

Управление исключениями

В современных библиотеках управление исключениями может достичь новых уровней сложности — в основном из-за недокументированных или плохо документированных подпрограмм обработки исключений. Каждый, кто знаком с современными операционными системами, наверняка сталкивался с ошибками и проблемами, которые связаны с необработанными исключениями.

При этом программистам часто приходится до изнеможения копаться в содержимом библиотек, разбираясь, какие исключения сгенерированы, когда и где они возникли и как их обработать.

Программисты-новички быстро уясняют, что изучение управления исключениями в библиотеках не проще изучения самого языка: современные библиотеки могут содержать программы и парадигмы, столь же непривычные и сложные, как и некоторые нюансы синтаксиса C++. Для построения качественных проектов понимание тонких мест библиотек и классов столь же важно, как и изучение самого языка C++. Освоение обработки ошибок и исключе- ний на основе документации и кода библиотеки принесет вам и вашим программам боль- шую пользу.

Динамическая идентификация типов

Динамическая идентификация типов (Run Time Type Identification — RTTI) — это одно из новейших расширений языка C++, и оно не поддерживается многими устаревшими реализациями. В некоторых реализациях имеются опции компилятора, позволяющие включать и отключать RTTI. Главная цель введения RTTI — предоставить программам стандартный механизм для определения типа объектов во время выполнения. Во многих библиотеках классов такой механизм уже предусмотрен применительно к собственным классам. Однако без встроенной в C++ поддержки механизмы разных поставщиков библиотек вряд ли будут совместимыми. Создание стандарта языка для RTTI позволит обеспечить совместимость будущих библиотек.

Для чего нужен механизм RTTI

Предположим, что существует иерархия классов, порожденных от общего базового класса. Указатель на базовый класс может содержать адрес объекта любого класса из этой иерархии. Можно вызвать функцию, которая, обработав какую-то информацию, выбирает один из существующих классов, создает объект этого типа и возвращает его адрес, который заносится в указатель на базовый класс. Как теперь определить, на какой тип объекта он указывает?

Прежде чем ответить на этот вопрос, нужно уяснить, зачем вообще знать тип. Возможно, требуется просто вызвать соответствующую версию метода класса. Но в таком случае знание типа объекта не требуется, т.к. эта функция является виртуальной и известна всем членам иерархии класса. Однако может оказаться, что производный объект имеет собственный, а не унаследованный метод. В этом случае его смогут использовать только некоторые объекты. Или для целей отладки может понадобиться узнать тип сгенерированного объекта. Для двух последних случаев ответ дает механизм RTTI.

Как работает механизм RTTI

В C++ есть три компонента, поддерживающие RTTI.

- Операция `dynamic_cast`, когда это возможно, создает указатель на производный класс из указателя на базовый класс. В случае невозможности операция возвращает `0`, т.е. нулевой указатель.
- Операция `typeid` возвращает величину, идентифицирующую точный тип объекта.
- Структура `type_info` содержит информацию о конкретном типе.

RTTI можно использовать только с иерархией классов, содержащей виртуальные функции. Причина в том, что это – единственная иерархия классов, для которой бывает нужно присваивать адреса производных объектов указателям базового класса.

Внимание!

RTTI работает только для классов, имеющих виртуальные функции.

Рассмотрим все три указанных компонента.

Операция dynamic_cast

Операция dynamic_cast – наиболее часто используемый компонент RTTI. Она не отвечает, на какой тип объекта указывает указатель. Вместо этого она дает ответ на вопрос, можно ли безопасно присвоить адрес объекта указателю на некоторый тип. Посмотрим, что это значит. Предположим, что существует следующая иерархия:

```
class Grand { // содержит виртуальные методы };
class Superb : public Grand { ... };
class Magnificent : public Superb { ... };
```

Далее, пусть имеются перечисленные ниже указатели:

```
Grand * pg = new Grand;
Grand * ps = new Superb;
Grand * pm = new Magnificent;
```

И рассмотрим следующие приведения типов:

```
Magnificent * p1 = (Magnificent *) pm;       // #1
Magnificent * p2 = (Magnificent *) pg;       // #2
Superb * p3 = (Magnificent *) pm;            // #3
```

Какие из этих приведений типов безопасны? В зависимости от объявлений классов, все они могут быть безопасными. Однако единственным гарантированно безопасным приведением является такое, в котором указатель имеет тот же тип, что и объект, либо базовый для этого объекта тип (непосредственный или более дальний предок). Например, приведение #1 безопасно, потому что оно заносит в указатель типа Magnificent адрес объекта типа Magnificent. Приведение типов #2 не безопасно: оно присваивает адрес объекта базового класса (Grand) указателю на производный класс (Magnificent). После этого программа будет ожидать, что объект базового класса содержит свойства производного класса, что в общем случае неверно. К примеру, объект Magnificent может содержать данные-члены, которых нет в классе Grand. А вот приведение #3 безопасно, т.к. в нем указателю на базовый класс присваивается адрес производного объекта. То есть открытое наследование обеспечивает, что объект Magnificent также является объектом Superb (непосредственный базовый класс) и объектом Grand (дальний предок). Поэтому допустимо присваивать его адрес указателям всех трех типов. Виртуальные функции гарантируют, что использование указателей всех трех типов с адресом Magnificent приведет к вызову методов Magnificent.

Обратите внимание, что вопрос, безопасно ли некоторое преобразование типов, является и более общим, и более важным, нежели вопрос, на какой тип объекта указывает указатель. Обычно знание типа объекта нужно тогда, когда требуется знать, безопасным ли будет вызов конкретного метода. Для вызова метода не обязательно иметь точное совпадение типа. Типом может быть базовый тип, для которого определена виртуальная версия метода. Это будет продемонстрировано в примере, показанном чуть ниже.

Но сначала рассмотрим синтаксис операции dynamic_cast. Она используется следующим образом (pg указывает на объект):

```
Superb * pm = dynamic_cast<Superb *>(pg);
```

Здесь спрашивается, можно ли тип указателя pg безопасно привести (как рассматривалось выше) к типу Superb *. Если да, операция возвращает адрес объекта. Иначе возвращается 0, т.е. нулевой указатель.

На заметку!

В общем случае выражение dynamic_cast<Type *>(pt) преобразует указатель pt в указатель типа Type *, если указываемый объект (*pt) имеет тип Type либо унаследован непосредственно или опосредованно от типа Type.

В противном случае это выражение вычисляется как 0, т.е. нулевой указатель.

Описанный процесс демонстрируется в листинге 15.17. Сначала создаются три класса — Grand, Superb и Magnificent. В классе Grand определяется виртуальная функция-член Speak(), которую все остальные классы переопределяют. В классе Superb() определяется виртуальная функция-член Say(), переопределяемая в классе Magnificent (рис. 15.4).

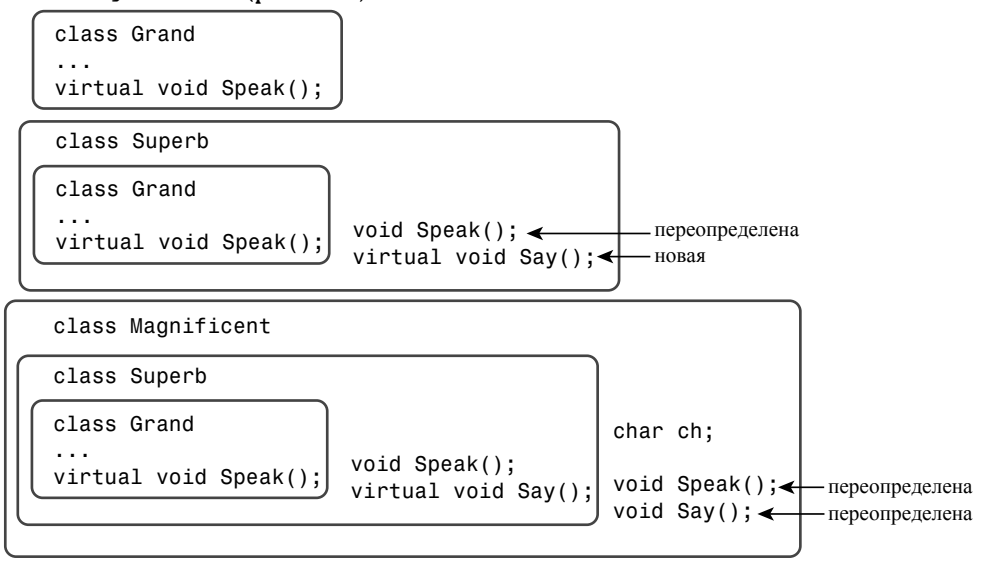


Рис. 15.4. Семейство классов Grand

В программе определена функция-член GetOne(), которая случайным образом создает и инициализирует объект одного из этих трех классов и возвращает адрес в виде указателя типа Grand *. (Функция GetOne() имитирует интерактивное принятие решения пользователем.) В цикле этот указатель присваивается переменной pg типа Garnd *, и затем pg используется для вызова функции Speak(). Поскольку эта функция виртуальная, код вызывает версию Speak(), соответствующую указываемому объекту:

```
for (int i = 0; i < 5; i++)
{
    pg = GetOne();
    pg->Speak();
    ...
}
```

Этот подход (с указателем на Grand) нельзя в точности использовать для вызова функции Say(): она не определена для класса Grand(). Однако операция dynamic_cast позволяет узнать, можно ли тип pg привести к указателю на Superb. Это возможно, если объект имеет тип Superb или Magnificent. В любом из этих случаев можно безопасно вызвать функцию Say():

```
if (ps = dynamic_cast<Superb *>(pg))
   ps->Say();
```

Вспомните, что значением в выражении присваивания является значение, которое присваивается левой части. Поэтому значение выражения if — это ps. Если приведение типа выполнится успешно, ps будет ненулевым, т.е. true. Если приведение типа завершится неудачно (если pg указывает на объект Grand), то ps будет равно нулю, или false. В листинге 15.17 приведен полный код. (Кстати, некоторые компиляторы, зная, что программисты обычно используют в условии if операцию ==, могут выдать предупреждение о ненамеренном присваивании.)

Листинг 15.17. rtti1.cpp

```
// rtti1.cpp -- использование RTTI-операции dynamic_cast
#include <iostream>
#include <cstdlib>
#include <ctime>
using std::cout;
class Grand
{
private:
   int hold;
public:
   Grand(int h = 0) : hold(h) {}
   virtual void Speak() const { cout << "I am a grand class!\n";}
   virtual int Value() const { return hold; }
};
class Superb : public Grand
{
public:
   Superb(int h = 0) : Grand(h) {}
   void Speak() const {cout << "I am a superb class!!\n"; }
   virtual void Say() const
      { cout << "I hold the superb value of " << Value() << "!\n";}
};
class Magnificent : public Superb
{
private:
   char ch;
public:
   Magnificent(int h = 0, char c = 'A') : Superb(h), ch(c) {}
   void Speak() const {cout << "I am a magnificent class!!!\n";}
   void Say() const {cout << "I hold the character " << ch <<
                             " and the integer " << Value() << "!\n"; }
};
Grand * GetOne();
int main()
{
   std::srand(std::time(0));
   Grand * pg;
   Superb * ps;
```

```
    for (int i = 0; i < 5; i++)
    {
        pg = GetOne();
        pg->Speak();
        if( ps = dynamic_cast<Superb *>(pg))
            ps->Say();
    }
    return 0;
}
Grand * GetOne() // случайным образом генерирует один из трех типов объектов
{
    Grand * p;
    switch( std::rand() % 3)
    {
        case 0: p = new Grand(std::rand() % 100);
                break;
        case 1: p = new Superb(std::rand() % 100);
                break;
        case 2: p = new Magnificent(std::rand() % 100,
                                'A' + std::rand() % 26);
                break;
    }
    return p;
}
```

На заметку!

Даже если ваш компилятор поддерживает RTTI, по умолчанию этот механизм может быть отключен. Если это так, то программа может нормально скомпилироваться, но приводить к ошибкам времени выполнения. Если вы столкнетесь с такой проблемой, обратитесь к документации.

Программа в листинге 15.17 иллюстрирует важную мысль. По возможности следует всегда использовать виртуальные функции, а RTTI – только когда это необходимо. Ниже показан пример вывода программы из листинга 15.17:

```
I am a superb class!!
I hold the superb value of 68!
I am a magnificent class!!!
I hold the character R and the integer 68!
I am a magnificent class!!!
I hold the character D and the integer 12!
I am a magnificent class!!!
I hold the character V and the integer 59!
I am a grand class!
```

Как видите, методы `Say()` вызваны только для классов `Superb` и `Magnificent`. (Вывод будет варьироваться от запуска к запуску, т.к. для выбора типа объекта в программе применяется функция `rand()`.)

Операция `dynamic_cast` применима и к ссылкам, хотя ее использование при этом слегка отличается. Поскольку не существует такого значения ссылки, которое соответствует типу нулевого указателя, невозможно выбрать специальное значение, которое обозначает неудавшееся выполнение. Вместо этого операция `dynamic_cast` генерирует исключение `bad_cast`, производное от класса `exception` и определенное в заголовочном файле `typeinfo`. Поэтому можно применить следующую операцию (`rg` – ссылка на объект `Grand`):

```
#include <typeinfo>  // для bad_cast
...
try {
   Superb & rs = dynamic_cast<Superb &>(rg);
   ...
}
catch(bad_cast &){
   ...
};
```

Операция typeid и класс type_info

Операция typeid позволяет выяснить, совпадают ли типы объектов. Похожая на sizeof, она принимает аргументы двух видов:

- имя класса;
- выражение, которое вычисляется как тип объекта.

Операция typeid возвращает ссылку на объект type_info, определенный в заголовочном файле typeinfo (ранее typeinfo.h). Класс type_info перегружает операции == и !=, чтобы их можно было использовать для сравнения типов. Например, следующее выражение возвращает булевское значение true, если pg указывает на объект Magnificent, и false — в противном случае:

```
typeid(Magnificent) == typeid(*pg)
```

Если pg окажется нулевым указателем, программа сгенерирует исключение bad_typeid. Этот тип исключения порожден от класса exception и определен в заголовочном файле typeinfo.

Реализация класса type_info отличается у разных разработчиков компиляторов, но она обязательно содержит член name(), который возвращает зависящую от реализации строку, обычно (но не обязательно) содержащую имя класса. Например, приведенный ниже оператор отображает строку, определенную для класса объекта, на который указывает указатель pg:

```
cout << "Now processing type " << typeid(*pg).name() << ".\n";
```

Код в листинге 15.18 представляет собой модифицированную версию кода из листинга 15.17: в нем используются операция typeid и функция-член name(). Они применяются в ситуациях, в которых операция dynamic_cast и виртуальные функции бессильны. Проверка typeid используется для выбора действия, которое даже не является методом класса, поэтому ее нельзя вызвать с помощью указателя на класс. Оператор с методом name() демонстрирует, как можно использовать этот метод для отладки. Обратите внимание, что в программе включается заголовочный файл typeinfo.

Листинг 15.18. rtti2.cpp

```
// rtti2.cpp -- использование dynamic_cast, typeid и type_info
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <typeinfo>
using namespace std;
class Grand
{
private:
   int hold;
```

```
public:
   Grand(int h = 0) : hold(h) {}
   virtual void Speak() const { cout << "I am a grand class!\n";}
   virtual int Value() const { return hold; }
};

class Superb : public Grand
{
public:
   Superb(int h = 0) : Grand(h) {}
   void Speak() const {cout << "I am a superb class!!\n"; }
   virtual void Say() const
      { cout << "I hold the superb value of " << Value() << "!\n";}
};

class Magnificent : public Superb
{
private:
   char ch;
public:
   Magnificent(int h = 0, char cv = 'A') : Superb(h), ch(cv) {}
   void Speak() const {cout << "I am a magnificent class!!!\n";}
   void Say() const {cout << "I hold the character " << ch <<
                             " and the integer " << Value() << "!\n"; }
};

Grand * GetOne();

int main()
{
   srand(time(0));
   Grand * pg;
   Superb * ps;
   for (int i = 0; i < 5; i++)
   {
      pg = GetOne();
      cout << "Now processing type " << typeid(*pg).name() << ".\n";
      pg->Speak();
      if( ps = dynamic_cast<Superb *>(pg))
         ps->Say();
      if (typeid(Magnificent) == typeid(*pg))
         cout << "Yes, you're really magnificent.\n";
   }
   return 0;
}

Grand * GetOne()
{
   Grand * p;
   switch( rand() % 3)
   {
      case 0: p = new Grand(rand() % 100);
              break;
      case 1: p = new Superb(rand() % 100);
              break;
      case 2: p = new Magnificent(rand() % 100, 'A' + rand() % 26);
              break;
   }
   return p;
}
```

Ниже показан пример запуска программы из листинга 15.18:

```
Now processing type Magnificent.
I am a magnificent class!!!
I hold the character P and the integer 52!
Yes, you're really magnificent.
Now processing type Superb.
I am a superb class!!
I hold the superb value of 37!
Now processing type Grand.
I am a grand class!
Now processing type Superb.
I am a superb class!!
I hold the superb value of 18!
Now processing type Grand.
I am a grand class!
```

Как и в предыдущем примере, вывод программы варьируется от запуска к запуску, поскольку для выбора типа объекта используется функция `rand()`. Кроме того, некоторые компиляторы могут выдавать различные результаты при вызове `name()` — например, `5Grand`, а не `Grand`.

Неправильное использование RTTI

В сообществе C++ есть много критиков RTTI. Они считают RTTI бесполезным добавлением и потенциальной причиной неэффективности программ и плохого стиля программирования. Не углубляясь в эти дебаты, рассмотрим несколько приемов программирования, которых следует избегать.

Взгляните на основную часть листинга 15.17:

```
Grand * pg;
Superb * ps;
for (int i = 0; i < 5; i++)
{
   pg = GetOne();
   pg->Speak();
   if (ps = dynamic_cast<Superb *>(pg))
      ps->Say();
}
```

Используя `typeid` и игнорируя `dynamic_cast` и виртуальные функции, этот код можно переписать так:

```
Grand * pg;
Superb * ps;
Magnificent * pm;
for (int i = 0; i < 5; i++)
{
   pg = GetOne();
   if (typeid(Magnificent) == typeid(*pg))
   {
      pm = (Magnificent *) pg;
      pm->Speak();
      pm->Say();
   }
   else if (typeid(Superb) == typeid(*pg))
   {
      ps = (Superb *) pg;
```

```
        ps->Speak();
        ps->Say();
    }
    else
        pg->Speak();
}
```

Полученный код не только корявей и длиннее исходного, но он еще имеет серьезный недостаток — явное именование каждого класса. Предположим, что от класса `Magnificent` требуется породить класс `Insufferable`, который переопределит методы `Speak()` и `Say()`. В версии с использованием `typeid` для явной проверки каждого типа придется добавить в цикл `for` новый раздел `else if`. А вот исходная версия не потребует изменений вообще. Следующий оператор работает для всех классов, порожденных от `Grand`:

```
pg->Speak();
```

А этот оператор работает для всех классов, порожденных от `Superb`:

```
if (ps = dynamic_cast<Superb *>(pg))
  ps->Say();
```

Совет

Если вы используете операцию `typeid` в длинной последовательности операторов `if else`, возможно, лучше будет задействовать виртуальные функции и операцию `dynamic_cast`.

Операции приведения типов

По мнению Бьярне Страуструпа, операция приведения типа в языке C слишком нестрогая. Например, рассмотрим следующий фрагмент:

```
struct Data
{
   double data[200];
};
struct Junk
{
   int junk[100];
};
Data d = {2.5e33, 3.5e-19, 20.2e32};
char * pch = (char *) (&d);     // приведение типа #1 — преобразование в строку
char ch = char (&d);            // приведение типа #2 — преобразование в символ
Junk * pj = (Junk *) (&d);      // приведение типа #3 — преобразование
                                // в указатель на Junk
```

Во-первых, какие из этих трех приведений имеют смысл? Ни одно из них, если находиться в здравом уме. Во-вторых, какие из этих приведений допустимы? В языке C все они являются допустимыми. Реакцией Страуструпа на такую вольность было четкое определение того, что является допустимым для общих приведений типов, и добавление четырех операций приведения типов, которые вводят некоторую дисциплину:

```
dynamic_cast
const_cast
static_cast
reinterpret_cast
```

Вместо обобщенного приведения типа можно использовать операцию, более подходящую для конкретной цели. Это позволяет указать на цель приведения и дает возможность компилятору проверить, делаете ли вы то, что намеревались.

Вы уже знакомы с операцией dynamic_cast. Пусть High и Low — два класса, переменная ph имеет тип High *, а pl — тип Low *. Тогда следующий оператор присваивает указатель Low * переменной pl, только если Low является доступным для High базовым классом (непосредственным или дальним предком):

```
pl = dynamic_cast<Low *> ph;
```

В противном случае этот оператор присваивает pl нулевой указатель. В общем случае операция dynamic_cast имеет такой синтаксис:

```
dynamic_cast <имя-типа> (выражение)
```

Назначение этой операции — разрешить восходящие приведения типа внутри иерархии классов (такие приведения будут безопасными в силу наличия отношения *является*) и запретить другие приведения.

Операция const_cast позволяет выполнить приведение типа, только если значение объявлено как const или volatile. Она имеет такой же синтаксис, как и dynamic_cast:

```
const_cast <имя-типа> (выражение)
```

При несовпадении любых других аспектов типов результатом такого приведения будет ошибка. То есть *имя-типа* и *выражение* должны быть одного типа; они могут отличаться только наличием или отсутствием квалификаторов const или volatile. Пусть опять High и Low — два класса, и рассмотрим следующий код:

```
High bar;
const High * pbar = &bar;
...
High * pb = const_cast<High *> (pbar);            // верно
const Low * pl = const_cast<const Low *> (pbar);  // неверно
```

Первое приведение типа делает *pb указателем, который можно использовать для изменения значения объекта bar; оно удаляет метку const. Второе приведение неверно, поскольку оно пытается изменить тип с const High * на const Low *.

Эта операция предназначена для того случая, когда нужно иметь величину, которая большую часть времени постоянна, но иногда все же может изменяться. В этом случае ее можно объявить как const и использовать операцию const_cast, если потребуется изменить ее значение. Это можно сделать с помощью обычного приведения типа, но тогда изменится и тип:

```
High bar;
const High * pbar = &bar;
...
High * pb = (High *) (pbar);   // верно
Low * pl = (Low *) (pbar);     // тоже верно
```

Поскольку одновременное изменение типа и постоянства значения может оказаться непреднамеренной программной ошибкой, безопаснее применять операцию const_cast.

Операция const_cast не во всем хороша. Она может изменить указатель на величину, но эффект попытки изменения значения, объявленного с квалификатором const, не определен. Для наглядности рассмотрим короткий пример, приведенный в листинге 15.19.

Листинг 15.19. constcast.cpp

```
// constcast.cpp -- использование const_cast<>
#include <iostream>
using std::cout;
using std::endl;
void change(const int * pt, int n);
int main()
{
    int pop1 = 38383;
    const int pop2 = 2000;
    cout << "pop1, pop2: " << pop1 << ", " << pop2 << endl;
    change(&pop1, -103);
    change(&pop2, -103);
    cout << "pop1, pop2: " << pop1 << ", " << pop2 << endl;
    return 0;
}
void change(const int * pt, int n)
{
    int * pc;
    pc = const_cast<int *>(pt);
    *pc += n;
}
```

Операция `const_cast` может удалить квалификатор `const` из `const int * pt`, что позволяет компилятору в функции `change()` воспринять следующий оператор:

```
*pc += n;
```

Но поскольку переменная `pop2` объявлена как `const`, компилятор может защитить ее от изменений, как показано в следующем примере выходных данных программы:

```
pop1, pop2: 38383, 2000
pop1, pop2: 38280, 2000
```

Как видите, вызов `change()` изменяет значение `pop1`, но не `pop2`. Указатель в функции `change()` объявлен как `const int *`, поэтому его нельзя использовать для изменения целого числа, на которое он указывает. Указатель `pc` отбрасывает приведение `const`, и он позволяет изменить значение, на которое указывает, но только если само значение не объявлено как `const`. Поэтому `pc` можно использовать для изменения `pop1`, но не `pop2`.

Операция `static_cast` имеет такой же синтаксис, как и другие операции:

```
static_cast <имя-типа> (выражение)
```

Она допустима только в том случае, если *имя-типа* может быть неявно преобразовано в тип, который имеет *выражение*, или наоборот. В любом другом случае такое приведение считается ошибочным. Пусть `High` — базовый класс для `Low`, а `Pond` — несвязанный класс. Тогда приведение `High` к `Low` и обратно допустимо, а приведение `Low` к `Pond` — нет:

```
High bar;
Low blow;
...
High * pb = static_cast<High *> (&blow);  // допустимое восходящее приведение
Low * pl = static_cast<Low *> (&bar);     // допустимое нисходящее приведение
Pond * pmer = static_cast<Pond *> (&blow); // не допускается, Pond не входит
                                           // в иерархию
```

Первое преобразование является допустимым, поскольку восходящее приведение может быть выполнено явно. Второе преобразование — из указателя на базовый класс в указатель на производный класс — не может быть выполнено без явного приведения типа. Но поскольку обратное преобразование возможно без приведения типа, операцию `static_cast` можно использовать для нисходящего преобразования.

Аналогично, значение перечисления может быть преобразовано в целый тип без приведения, и поэтому целочисленный тип можно преобразовать в значение перечисления с помощью операции `static_cast`. Кроме того, `static_cast` позволяет преобразовать `double` в `int`, `float` в `long` и выполнять многие другие числовые преобразования.

Операция `reinterpret_cast` предназначена для рискованных приведений типов. Она не позволит отбросить квалификатор `const`, но может вызвать другие неприятные вещи. Иногда программистам приходится писать зависящий от реализации код, и применение `reinterpret_cast` упрощает такой процесс. Эта операция имеет такой же синтаксис, что и остальные три операции:

```
reinterpret_cast <имя-типа> (выражение)
```

Ниже показан пример ее использования:

```
struct dat {short a; short b;};
long value = 0xA224B118;
dat * pd = reinterpret_cast<dat *> (&value);
cout << hex << pd->a;   // вывод первых двух байтов значения
```

Обычно такое приведение типа применяется при низкоуровневом, зависящем от реализации программировании, и полученный код не всегда возможно перенести в другую систему: эта система может хранить байты не в том формате или в другом порядке.

Однако операция `reinterpret_cast` не позволяет делать вообще все что угодно. Например, можно привести тип указателя к целочисленному типу, который достаточно велик, чтобы хранить указатель, но нельзя привести указатель к меньшему целому типу или к типу с плавающей точкой. Существует еще одно ограничение: нельзя привести указатель на функцию к указателю на данные и наоборот.

Обычное приведение типов в C++ также ограничено. В основном оно может делать то же, что и другие приведения, плюс сочетания вроде `static_cast` или `reinterpret_cast`, за которыми следует `const_cast`. Но это, пожалуй, и все. Так что следующее приведение допустимо в C, но обычно запрещено в C++, потому что в большинстве реализаций C++ тип `char` слишком мал, чтобы содержать указатель:

```
char ch = char (&d);  // приведение типа #2 — преобразование адреса в символ
```

Такие меры предосторожности имеют смысл, но если вы считаете их излишними — к вашим услугам язык C.

Резюме

Друзья позволяют разрабатывать для классов более гибкий интерфейс. Друзьями класса могут быть другие функции, другие классы и функции-члены других классов. Для эффективного сосуществования друзей в некоторых случаях требуются предварительные объявления и аккуратность в расположении объявлений классов и методов.

Вложенными классами называются классы, которые объявлены внутри других классов. Вложенные классы упрощают создание вспомогательных классов, которые реализуют другие классы, но не являются частью открытого интерфейса.

Механизм исключений в C++ предоставляет гибкое средство для работы с нежелательными программными событиями, такими как недопустимые значения, неудачные попытки ввода-вывода и т.п. Генерация исключения прерывает выполнение текущей функции и передает управление в соответствующий блок `catch`, который находится сразу за блоком `try`. Чтобы исключение было перехвачено, вызов функции, который прямо или косвенно приводит к возникновению этого исключения, должен находиться в блоке `try`.

После этого программа выполняет блок `catch`. Этот код может попытаться устранить проблему или прекратить выполнение программы. Можно создать класс с вложенными классами исключений, которые генерируется при возникновении соответствующих проблем. Функция может содержать спецификацию исключений, определяющую исключения, которые могут быть сгенерированы в этой функции, хотя в C++11 это средство объявлено устаревшим. Неперехваченные исключения (не имеющие соответствующего блока `catch`) по умолчанию завершают выполнение программы. Завершают программу и непредвиденные исключения (не указанные в спецификации исключений).

Компоненты RTTI позволяют программам определять типы объектов. Операция `dynamic_cast` используется для приведения указателя на производный класс к указателю на базовый класс. Ее главная цель – убедиться, что все нормально, при вызове виртуальной функции. Операция `typeid` возвращает объект `type_info`. Для определения, имеет ли объект определенный тип, можно сравнить два значения, возвращенные `typeid`. Возвращенный объект `type_info` позволяет также получить информацию о самом объекте.

Операции `dynamic_cast`, `static_cast`, `const_cast` и `reinterpret_cast` предоставляют более надежный и лучше документированный механизм приведения типов по сравнению с общим механизмом приведения.

Вопросы для самоконтроля

1. Что неверно в следующих попытках создания дружественных конструкций:

а.
```
class snap {
    friend clasp;
    ...
};
class clasp { ... };
```

б.
```
class cuff {
public:
    void snip(muff &) { ... }
    ...
};
class muff {
    friend void cuff::snip(muff &);
    ...
};
```

в.
```
class muff {
    friend void cuff::snip(muff &);
    ...
};
class cuff {
public:
    void snip(muff &) { ... }
    ...
};
```

2. Вы уже видели, как создаются взаимно дружественные классы. Возможно ли создать более ограниченную форму отношения дружественности, при котором только некоторые члены класса B являются друзьями для класса A и некоторые члены класса A — друзьями для B? Обоснуйте свой ответ.

3. Какие проблемы могут возникнуть в следующем объявлении вложенного класса?

```
class Ribs
{
private:
   class Sauce
   {
      int soy;
      int sugar;
   public:
      Sauce(int s1, int s2) : soy(s1), sugar(s2) { }
   };
   ...
};
```

4. В чем состоит различие между `throw` и `return`?

5. Предположим, что имеется иерархия классов исключений, порожденная от базового класса исключений. В каком порядке следует расположить блоки `catch`?

6. Рассмотрим классы `Grand`, `Superb` и `Magnificent`, определенные в настоящей главе. Пусть `pg` — указатель типа `Grand *`, которому присвоен адрес объекта одного из этих трех классов, а `ps` — указатель типа `Superb *`. В чем разница в поведении двух следующих примеров кода?

```
if (ps = dynamic_cast<Superb *>(pg))
   ps->say();                  // пример #1
if (typeid(*pg) == typeid(Superb))
   (Superb *) pg)->say();  // пример #2
```

7. Чем отличается операция `static_cast` от операции `dynamic_cast`?

Упражнения по программированию

1. Измените классы `Tv` и `Remote`, как описано ниже.

   а. Сделайте их взаимными друзьями.

   б. Добавьте в класс `Remote` переменную-член, описывающую режим пульта дистанционного управления — нормальный или интерактивный.

   в. Добавьте метод `Remote`, который отображает режим.

   г. Добавьте в класс `Tv` метод для переключения нового члена `Remote`. Этот метод должен работать, только если телевизор включен.

   Напишите небольшую программу для тестирования новых возможностей.

2. Измените код в листинге 15.11 так, чтобы два типа исключений были классами, производными от класса `logic_error`, определенного в заголовочном файле `<stdexcept>`. Сделайте так, чтобы каждый метод `what()` сообщал имя функции и суть проблемы. Объекты исключений не должны содержать значение ошибки, они должны просто поддерживать метод `what()`.

3. Это упражнение подобно упражнению 2, но исключения должны быть производными от базового класса (потомка `logic_error`), который хранит два значения аргументов. Исключения должны содержать метод, который выводит эти значения и имя функции, и единственный блок `catch`, который используется для обоих исключений. Каждое исключение должно приводить к прекращению цикла обработки.
4. В листинге 15.16 после каждого блока `try` находятся два блока `catch`, поэтому исключение `nbad_index` приводит к вызову метода `label_val()`. Измените программу так, чтобы она содержала один блок `catch` после каждого блока `try` и использовала RTTI для вызова `label_val()` лишь тогда, когда это необходимо.

16

Класс `string` и стандартная библиотека шаблонов

В ЭТОЙ ГЛАВЕ...

- Стандартный класс `string` в C++
- Шаблоны `auto_ptr`, `unique_ptr` и `shared_ptr`
- Стандартная библиотека шаблонов (STL)
- Классы контейнеров
- Итераторы
- Объекты функций (функторы)
- Алгоритмы STL
- Шаблон `initializer_list`

Теперь, после изучения предыдущих глав, идея повторного использования кода в C++ должна стать более понятной. Одно из главных преимуществ – повторное использование кода, написанного другими программистами. Именно здесь на первый план выходят библиотеки классов. Существует множество библиотек классов C++, как отдельно распространяемых на коммерческой основе, так и поставляемых в составе пакета C++. Например, мы уже использовали классы ввода-вывода, поддерживаемые заголовочным файлом `ostream`. В этой главе рассмотрены другие виды кода, доступные для повторного использования в программах.

В данной книге вы уже сталкивались с классом `string`. В настоящей главе он будет рассмотрен более подробно. Затем будут описаны классы шаблонов "интеллектуальных указателей", которые несколько облегчают управление динамической памятью. Потом мы рассмотрим стандартную библиотеку шаблонов (Standard Template Library – STL), которая представляет собой коллекцию полезных шаблонов, предназначенных для работы с разнообразными контейнерными объектами. Библиотека STL иллюстрирует популярную идеологию программирования – обобщенное программирование. И, наконец, будет представлен класс шаблона `initializer_list` – дополнение C++11, которое позволяет использовать синтаксис списка инициализаторов с объектами STL.

Класс `string`

Обработка строк требуется во многих приложениях. С помощью своего семейства функций работы со стоками `string.h` (`cstring` в C++) язык C предлагает некоторую поддержку решения этих задач, а многие ранние реализации C++ предоставляют собственные классы для обработки строк. Класс `string` из ANSI/ISO C++ был представлен в главе 4. Умеренно сложный класс `String`, описанный в главе 12, иллюстрирует некоторые аспекты разработки класса для представления строк.

Вспомните, что класс `string` поддерживается заголовочным файлом `string`. (Следует отметить, что заголовочные файлы `string.h` и `cstring` предлагают библиотеку функций для работы со строками в стиле языка C, но не класс `string`.) Для использования класса нужно знать предоставляемый им открытый интерфейс. В классе `string` определен обширный набор методов для работы со строками. В этот набор входят несколько конструкторов, перегруженных операций для доступа к строкам, конкатенации и сравнения строк, доступа к отдельным элементам строки, а также средства поиска символов и подстрок в строке. И это далеко не полный перечень возможностей класса `string`.

Создание объекта `string`

Рассмотрим конструкторы класса `string`. Одна из самых важных вещей, которые нужно знать о классе – какие варианты доступны при создании его объектов. В листинге 16.1 использованы семь конструкторов класса `string` (они помечены комментариями `ctor` – стандартной аббревиатурой *конструктора* в C++). Краткое описание конструкторов приведено в табл. 16.1. Таблица начинается с описаний семи конструкторов, использованных в листинге 16.1 (в порядке их применения). Кроме того, она содержит несколько конструкторов, добавленных в C++11. Представления конструкторов упрощены в том смысле, что они скрывают тот факт, что на самом деле `string` – это `typedef` для специализации шаблона `basic_string<char>`, и в них опущен необязательный параметр, относящийся к управлению памятью. Этот аспект рассматривается далее в этой главе и в приложении E. `size_type` является внутренним, зависящим от реализации типом, который определен в заголовочном файле `string`.

Класс определяет `string::npos` в качестве максимально возможной длины строки. Как правило, его значение будет равно максимальному значению типа `unsigned int`.

Таблица 16.1. Конструкторы класса `string`

Конструктор	Описание
`string(const char * s)`	Инициализирует объект `string` строкой, завершающейся нулевым байтом, которая указана в *s*
`string(size_type n, char c)`	Создает объект типа `string` из *n* элементов, каждый из которых инициализируется символом *c*
`string(const string & str)`	Инициализирует объект `string` объектом *str* типа `string` (конструктор копирования)
`string()`	Создает объект типа `string` нулевого размера (конструктор по умолчанию)
`string(const char * s, size_type n)`	Инициализирует объект типа `string` строкой, завершающейся нулевым байтом, которая указана в *s* и содержит *n* символов, даже если *n* превышает длину *s*
`template<class Iter> string(Iter begin, Iter end)`	Инициализирует объект типа `string` значениями в диапазоне [*begin*, *end*), причем *begin* и *end* служат указателями начала и конца диапазона. Диапазон начинается с позиции *begin* включительно и заканчивается позицией *end*, не включая ее
`string(const string & str, size_type pos, size_type n = npos)`	Инициализирует объект типа `string` объектом *str*, начиная с позиции *pos* и оканчивая концом *str* либо ограничиваясь *n* символами, в зависимости от того, какое условие будет удовлетворено раньше
`string(string && str) noexcept` (C++11)	Инициализирует объект `string` объектом *str* типа `string`. Объект *str* может быть изменен (конструктор переноса)
`string(initializer_list<char> il)` (C++11)	Инициализирует объект типа `string` символами, указанными в списке инициализации *il*

Листинг 16.1. `str1.cpp`

```
// str1.cpp -- введение в класс string
#include <iostream>
#include <string>

// Использование различных конструкторов класса string
int main()
{
   using namespace std;
   string one("Lottery Winner!");        // ctor #1
   cout << one << endl;                  // перегруженная <<
   string two(20, '$');                  // ctor #2
   cout << two << endl;
   string three(one);                    // ctor #3
   cout << three << endl;
   one += " Oops!";                      // перегруженная +=
   cout << one << endl;
   two = "Sorry! That was ";
   three[0] = 'P';
```

```
    string four;                              // ctor #4
    four = two + three;                       // перегруженная +,=
    cout << four << endl;
    char alls[] = "All's well that ends well";
    string five(alls,20);                     // ctor #5
    cout << five << "!\n";
    string six(alls+6, alls + 10              // ctor #6
    cout << six << ", ";
    string seven(&five[6], &five[10]);        // снова ctor #6
    cout << seven << "...\n";
    string eight(four, 7, 16);                // ctor #7
    cout << eight << " in motion!" << endl;
    return 0;
}
```

В программе из листинга 16.1 также используется перегруженная операция += для добавления строк, перегруженная операция = для присваивания одной переменной типа `string` другой, перегруженная операция << для отображения объекта `string` и перегруженная операция `[]` для доступа к отдельным символам в строке.

Вывод этой программы имеет следующий вид:

```
Lottery Winner!
$$$$$$$$$$$$$$$$$$$$
Lottery Winner!
Lottery Winner! Oops!
Sorry! That was Pottery Winner!
All's well that ends!
well, well...
That was Pottery in motion!
```

Замечания по программе

Начало программы в листинге 16.1 иллюстрирует возможность инициализации объекта `string` обычной строкой в стиле C и ее вывод на экран с помощью перегруженной операции <<:

```
string one("Lottery Winner!");          // ctor #1
cout << one << endl;                    // перегруженная <<
```

Следующий конструктор инициализирует объект `two` типа `string` строкой, состоящей из 20 символов `$`:

```
string two(20, '$');                    // ctor #2
```

Конструктор копирования инициализирует объект `three` типа `string` объектом `one` этого же типа:

```
string three(one);                      // ctor #3
```

Перегруженная операция += дописывает строку `" Oops!"` к строке `one`:

```
one += " Oops!";                        // перегруженная операция +=
```

В этом примере строка в стиле языка C добавляется к объекту типа `string`. Однако операция += имеет несколько перегрузок и с ее помощью можно добавлять как объекты `string`, так и отдельные символы:

```
one += two;   // добавляет объект типа string (в программе этой строки нет)
one += '!';   // добавляет значение типа char (в программе этой строки нет)
```

Аналогично, операция = тоже является перегружаемой, что позволяет присвоить объекту типа string другой объект этого же типа, строку в стиле C или простое значение типа char:

```
two = "Sorry! That was ";      // присваивание строки в стиле C
two = one;          // присваивание объекта string (в программе этой строки нет)
two = '?';          // присваивание значения char (в программе этой строки нет)
```

Перегрузка операции [], как показано в примере класса String из главы 12, позволяет обращаться к отдельным символам объекта типа string, используя нотацию массива:

```
three[0] = 'P';
```

Конструктор по умолчанию создает пустую строку, которой впоследствии может быть присвоено значение:

```
string four;                    // ctor #4
four = two + three;             // перегруженные операции + и =
```

Во второй строке перегруженная операция + используется для создания временного объекта string, который затем с помощью перегруженной операции = присваивается объекту four. Операция + объединяет два операнда в один объект типа string. Эта операция имеет несколько перегрузок, поэтому второй операнд может быть объектом типа string, строкой в стиле C или значением типа char.

Пятый конструктор принимает в качестве аргументов строку в стиле C и целочисленное значение, которое указывает количество копируемых символов:

```
char alls[] = "All's well that ends well";
string five(alls,20);           // ctor #5
```

В выводе программы видно, что для инициализации объекта five использованы только первые 20 символов ("All's well that ends"). Как было указано в табл. 16.1, если количество символов превышает длину строки в стиле C, запрошенное количество символов все равно копируется. Поэтому замена значения 20 значением 40 в приведенном примере привела бы к копированию в конец строки five пятнадцати бессмысленных символов. (То есть конструктор интерпретировал бы содержимое памяти, следующей за строкой "All's well that ends well", как коды символов.) Шестой конструктор использует шаблон в качестве аргумента:

```
template<class Iter> string(Iter begin, Iter end);
```

begin и *end* выступают в роли указателей на начало и конец диапазона памяти. (В общем случае *begin* и *end* могут быть итераторами — обобщениями указателей, которые активно используются в STL.) Конструктор применяет значения элементов памяти, хранящиеся между позициями, указанными аргументами *begin* и *end*, для инициализации создаваемого им объекта типа string. Запись [*begin*, *end*), заимствованная из математики, означает, что диапазон включает в себя *begin*, но не включает *end*.

Другими словами, *end* указывает на позицию, следующую за последним значением, которое должно быть использовано. Рассмотрим следующий оператор:

```
string six(alls+6, alls + 10);   // ctor #6
```

Поскольку имя массива является указателем, значения alls + 6 и alls + 10 будут иметь тип char *, и поэтому в шаблоне тип Iter заменяется типом char *. Первый аргумент указывает на первый элемент (w) в массиве alls, а второй аргумент — на пробел после первого слова well. В результате объект six инициализируется строкой "well". Работа конструктора продемонстрирована на рис. 16.1.

```
char alls[] = "All's well that ends well";
string six(alls + 6, alls + 10);
  range = [alls + 6, alls + 10)
```

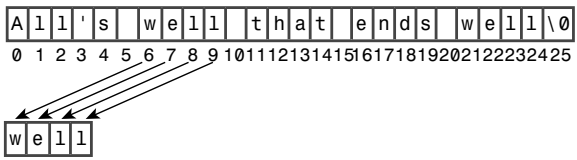


***Рис. 16.1.** Конструктор объекта `string`, использующий диапазон*

Предположим, что нужно инициализировать объект частью другого объекта типа `string`, например, объекта `five`. Следующий код работать не будет:

```
string seven(five + 6, five + 10);
```

Причина в том, что имя объекта, в отличие от имени массива, не является адресом объекта. Следовательно, `five` — не указатель, и выражение `five + 6` не имеет смысла. Однако `five[6]` является значением типа `char`, поэтому выражение `&five[6]` — это адрес, который может использоваться в качестве аргумента конструктора:

```
string seven(&five[6], &five[10]);      // снова ctor #6
```

Седьмой конструктор копирует часть объекта типа `string` в созданный объект:

```
string eight(four, 7, 16);              // ctor #7
```

Этот оператор копирует 16 символов из объекта `four` в объект `eight`, начиная с седьмой позиции (восьмого символа) объекта `four`.

Конструкторы C++11

Конструктор `string(string && str) noexcept` подобен конструктору копирования в том смысле, что новый объект `string` является копией объекта `str`. Однако, в отличие от конструктора копии, он не гарантирует, что объект `str` будет трактоваться как `const`. Эту форму конструктора называют *конструктором переноса*. В некоторых ситуациях компилятор может использовать его вместо конструктора копирования для оптимизации производительности. Этот вопрос рассмотрен в разделе "Семантика переноса и ссылка `rvalue`" главы 18.

Конструктор `string(initializer_list<char> il)` обеспечивает возможность списковой инициализации класса `string`. То есть он делает возможными объявления наподобие следующих:

```
string piano_man = {'L', 'i', 's','z','t'};
string comp_lang {'L', 'i', 's', 'p'};
```

Возможно, это не столь уж полезно для класса `string`, поскольку использование строк в стиле C проще, но позволяет сделать синтаксис списковой инициализации универсальным. Шаблон `initializer_list` будет рассмотрен далее в этой главе.

Ввод для класса `string`

Еще один аспект класса, который следует знать — доступные для него варианты ввода. Вспомните, что для строк в стиле C существуют три варианта ввода данных:

```
char info[100];
cin >> info;              // чтение слова
cin.getline(info, 100);   // чтение строки с отбрасыванием символа \n
cin.get(info, 100);       // чтение строки с сохранением символа \n в очереди
```

Для объектов string доступны два варианта:

```
string stuff;
cin >> stuff;                    // чтение слова
getline(cin, stuff);             // чтение строки с отбрасыванием символа \n
```

Обе версии вызова функции getline() допускают использование необязательного аргумента — символа, завершающего ввод:

```
cin.getline(info,100,':');       // чтение до символа ':', ':' отбрасывается
getline(stuff, ':');             // чтение до символа ':', ':' отбрасывается
```

Основное различие состоит в том, что версии с использованием string автоматически изменяют размер целевого объекта string в соответствии с количеством вводимых символов:

```
char fname[10];
string lname;
cin >> fname;                    // возможно возникновение проблем,
                                 // если вводится больше 9 символов
cin >> lname;                    // можно читать очень длинные строки
cin.getline(fname, 10);          // вводимая строка может быть усечена
getline(cin, fname);             // никакого усечения не выполняется
```

Функция автоматического определения размера позволяет версии getline(), использующей объект string, отказаться от параметра, который ограничивает количество вводимых символов.

Конструктивное же различие состоит в том, что средствами ввода строк в стиле C являются методы класса istream, а средствами версий с объектами string — автономные функции. Именно поэтому cin является вызывающим объектом для ввода строки в стиле C и аргументом функции для объекта ввода string. Это относится и к форме >>, что явно видно при записи кода в виде вызова функций:

```
cin.operator>>(fname);           // метод класса ostream
operator>>(cin, lname);          // обычный вызов функции
```

Рассмотрим работу функций ввода объектов string подробнее. Как упоминалось ранее, обе функции устанавливают размер целевой строки так, чтобы в нее уместились вводимые данные. Существует несколько ограничений на длину строки. Первый ограничивающий фактор — максимально допустимая длина строки, задаваемая константой string::npos. Обычно она равна максимальному значению типа unsigned int, и на практике этого хватает для обычного интерактивного ввода. Однако проблемы могут возникнуть при попытке считывания содержимого всего файла в один объект типа string. Второй ограничивающий фактор — размер памяти, доступной программе.

Функция getline(), применяемая к классу string, будет читать данные из входного потока и сохранять их в объекте string до тех пор, пока не произойдет одно из трех событий.

- Будет достигнут конец файла. В этом случае во входном потоке будет установлен флаг eofbit и обе функции fail() и eof() возвратят значение true.
- Будет достигнут разделительный символ (\n по умолчанию). Этот символ удаляется из входного потока, но не сохраняется.
- Будет прочитано максимально возможное количество символов (меньшее из значений константы string::npos и доступного количества байтов памяти). В этом случае во входном потоке будет установлен флаг failbit и функция fail() возвратит значение true.

(В объекте, обеспечивающем работу с входным потоком, имеется система учета для отслеживания ошибок при работе с потоком. В этой системе установленный флаг `eofbit` означает достижение конца файла, `failbit` — ошибку при чтении из потока, `badbit` — нераспознанный, например аппаратный, сбой и `goodbit` — нормальную работу без ошибок. Подробнее эти вопросы обсуждаются в главе 17.)

Функция `operator>>()` для класса `string` ведет себя аналогичным образом, за исключением того, что вместо чтения вплоть до разделительного символа и его отбрасывания она считывает данные из потока до тех пор, пока не встретится символ пробела, и оставляет его в очереди ввода. Символом пробела является собственно пробел, символ новой строки или символ табуляции, либо в более общем случае — любой символ, для которого функция `isspace()` возвращает значение `true`.

В книге уже приводились примеры консольного ввода для `string`. Поскольку функции ввода для объектов `string` работают с потоками и распознают конец файла, их можно применять для ввода из файла. Краткий пример считывания строк из файла приведен в листинге 16.2. В нем предполагается, что файл содержит строки, разделенные двоеточиями, и для указания этого разделителя использован метод `getline()`. Программа нумерует строки и выводит их на экран.

Листинг 16.2. `strfile.cpp`

```
// strfile.cpp -- чтение строк из файла
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>
int main()
{
   using namespace std;
   ifstream fin;
   fin.open("tobuy.txt");
   if (fin.is_open() == false)
   {
      cerr << "Can't open file. Bye.\n";     // не удается открыть файл
      exit(EXIT_FAILURE);
   }
   string item;
   int count = 0;
   getline(fin, item, ':');
   while (fin)                               // до тех пор, пока нет ошибок ввода
   {
      ++count;
      cout << count <<": " << item << endl;
      getline(fin, item,':');
   }
   cout << "Done\n";
   fin.close();
   return 0;
}
```

Предположим, что на вход программы подается файл `tobuy.txt`:

```
sardines:chocolate ice cream:pop corn:leeks:
cottage cheese:olive oil:butter:tofu:
```

Обычно программа будет искать текстовый файл в той папке, в которой находится исполняемый файл, либо, в ряде случаев, в папке, где расположен файл проекта.

Или же можно задать полный путь к файлу. Имейте в виду, что в системе Windows последовательность символов \\ в строке стиля C интерпретируется как один символ обратной черты \:

```
fin.open("C:\\CPP\\Progs\\tobuy.txt");  // file = C:\CPP\Progs\tobuy.txt
```

Вывод этой программы приведен в листинге 16.2.

```
1: sardines
2: chocolate ice cream
3: pop corn
4: leeks
5:
cottage cheese
6: olive oil
7: butter
8: tofu
9:

Done
```

Обратите внимание, что при использовании символа : в качестве разделителя символ перевода строки стал просто еще одним обычным символом. Поэтому символ перевода строки в конце первой строки файла стал первым символом строки, которая начинается со слов "cottage cheese". Аналогично символ перевода строки в конце второй строки ввода, если он присутствует, становится единственным элементом девятой строки.

Работа со строками

К этому моменту были рассмотрены различные способы создания объектов string, отображение их содержимого, считывание данных в объект, добавление к нему, присваивание значений объекту string и объединение двух объектов string. Что же еще можно делать со строками?

Строки можно сравнивать. Все шесть операций отношения перегружены для объектов string, причем один объект будет считаться меньше другого, если он находится раньше в машинной последовательности сопоставления. Если в основе последовательности сопоставления лежит код ASCII, то цифры будут считаться меньше прописных символов, а прописные символы — меньше строчных. Каждая операция отношения имеет три перегрузки, так что можно сравнивать объект string с другим объектом string, объект string со строкой в стиле C и строку в стиле C с объектом string:

```
string  snake1 ( " cobra " ) ;
string snake2("coral");
char snake3[20] = "anaconda";
if (snake1 < snake 2)          // operator<(const string &, const string &)
if (snake1 == snake3)          // operator==(const string &, const char *)
if (snake3 != snake2)          // operator!=(const char *, const string &)
```

Существуют две функции-члена класса string для определения размера строки — size() и length(), которые возвращают количество символов в строке:

```
if (snake1.length() == snake2.size())
   cout << "Both strings have the same length.\n";
         // Строки имеют одинаковую длину
```

Зачем же потребовалось две функции, которые делают одно и то же? Функция `length()` пришла из ранних версий класса `string`, а `size()` добавлена для совместимости с STL.

Поиск подстроки или символа в строке можно провести несколькими способами. В табл. 16.2 кратко описаны четыре варианта метода `find()`. Вспомните, что `string::npos` — максимально возможное количество символов в строке. Как правило, этим значением является наибольшее значение типа `unsigned int` или `unsigned long`.

Таблица 16.2. Перегруженный метод `find()`

Прототип метода	Описание
`size_type find(const string & str, size_type pos = 0) const`	Ищет в вызывающей строке первое вхождение подстроки *str*, начиная с позиции *pos*. Возвращает индекс первого символа найденной подстроки или `string::npos`, если подстрока не найдена
`size_type find(const char * s, size_type pos = 0) const`	Ищет в вызывающей строке первое вхождения подстроки *str*, начиная с позиции *pos*. Возвращает индекс первого символа найденной подстроки или `string::npos`, если подстрока не найдена
`size_type find(const char * s, size_type pos = 0, size_type n)`	Ищет в вызывающей строке первое вхождение подстроки, состоящей из первых *n* символов строки *s*, начиная с позиции *pos*. Возвращает индекс первого символа найденной подстроки или `string::npos`, если подстрока не найдена
`size_type find(char ch, size_type pos = 0) const`	Ищет в исходной строке первое вхождение символа *ch*, начиная с позиции *pos*. Возвращает индекс первого символа найденной подстроки или `string::npos`, если подстрока не найдена

Библиотека `string` также предоставляет связанные методы `rfind()`, `find_first_of()`, `find_last_of()`, `find_first_not_of()` и `find_last_not_of()`, каждый из которых имеет тот же набор сигнатур перегруженных функций, что и метод `find()`. Метод `rfind()` находит последнее вхождение подстроки или символа.

Метод `find_first_of()` отыскивает первое вхождение в строке любого из символов, переданных в аргументах метода. Например, следующий оператор вернет позицию символа `r` в строке `"cobra"` (3), поскольку это первое вхождение любого из символов строки `"hark"` в строке `"cobra"`:

```
int where = snake1.find_first_of("hark");
```

Метод `find_last_of()` работает аналогично, только находит последнее вхождение. Поэтому следующий оператор вернет позицию символа `a` в строке `"cobra"`:

```
int where = snake1.last_first_of("hark");
```

Метод `find_first_not_of()` находит первый символ в вызывающей строке, который отличается от символа, переданного в аргументе. Таким образом, приведенный ниже оператор вернет позицию символа `c` в `cobra`, поскольку символ `c` не найден в `hark`:

```
int where = snake1.find_first_not_of("hark");
```

(Примеры использования `find_last_not_of()` будут приведены в упражнениях в конце этой главы.)

Существует множество других методов, однако описанных выше достаточно для создания игровой программы — упрощенной версии игры "Палач" (детская игра в слова; при неправильном ответе игрок рисует одну за другой части виселицы с повешенным). Программа хранит список слов в массиве объектов `string`, выбирает одно слово случайным образом и предлагает игроку угадать буквы в слове. Шесть неудачных попыток означают проигрыш. В программе используется функция `find()` для проверки попыток и операция += для создания объекта `string`, в котором хранятся неудачные попытки. Для отслеживания удачных попыток, в программе создается слово такой же длины, что и загаданное, но состоящее из дефисов. При удачной попытке дефис заменяется угаданной буквой. Код программы приведен в листинге 16.3.

Листинг 16.3. `hangman.cpp`

```
// hangman.cpp -- использование некоторых методов работы со строками
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <cctype>
using std::string;
const int NUM = 26;
const string wordlist[NUM] = {"apiary", "beetle", "cereal",
   "danger", "ensign", "florid", "garage", "health", "insult",
   "jackal", "keeper", "loaner", "manage", "nonce", "onset",
   "plaid", "quilt", "remote", "stolid", "train", "useful",
   "valid", "whence", "xenon", "yearn", "zippy"};

int main()
{
   using std::cout;
   using std::cin;
   using std::tolower;
   using std::endl;
   std::srand(std::time(0));
   char play;
   cout << "Will you play a word game? <y/n> ";        // запуск игры в слова
   cin >> play;
   play = tolower(play);
   while (play == 'y')
   {
      string target = wordlist[std::rand() % NUM];
      int length = target.length();
      string attempt(length, '-');
      string badchars;
      int guesses = 6;
      cout << "Guess my secret word. It has " << length
           << " letters, and you guess\n"
           << "one letter at a time. You get " << guesses
           << " wrong guesses.\n";
      cout << "Your word: " << attempt << endl;        // вывод слова
      while (guesses > 0 && attempt != target)
      {
         char letter;
         cout << "Guess a letter: ";
         cin >> letter;
         if (badchars.find(letter) != string::npos
            || attempt.find(letter) != string::npos)
```

```
        {
            cout << "You already guessed that. Try again.\n";
            continue;
        }
        int loc = target.find(letter);
        if (loc == string::npos)
        {
            cout << "Oh, bad guess!\n";
            --guesses;
            badchars += letter;                    // добавить к строке
        }
        else
        {
            cout << "Good guess!\n";
            attempt[loc]=letter;
            // Проверить, не появляется ли буква еще раз
            loc = target.find(letter, loc + 1);
            while (loc != string::npos)
            {
                attempt[loc]=letter;
                loc = target.find(letter, loc + 1);
            }
        }
        cout << "Your word: " << attempt << endl;
        if (attempt != target)
        {
            if (badchars.length() > 0)
            cout << "Bad choices: " << badchars << endl;
            cout << guesses << " bad guesses left\n";
        }
    }
    if (guesses > 0)
      cout << "That's right!\n";
    else
      cout << "Sorry, the word is " << target << ".\n";
    cout << "Will you play another? <y/n> ";
    cin >> play;
    play = tolower(play);
  }
  cout << "Bye\n";
  return 0;
}
```

Ниже показа пример запуска программы из листинга 16.3:

```
Will you play a word game? <y/n> y
Guess my secret word. It has 6 letters, and you guess
one letter at a time. You get 6 wrong guesses.
Your word: ------
Guess a letter: e
Oh, bad guess!
Your word: ------
Bad choices: e
5 bad guesses left
Guess a letter: a
Good guess!
Your word: a--a--
```

```
Bad choices: e
5 bad guesses left
Guess a letter: t
Oh, bad guess!
Your word: a--a--
Bad choices: et
4 bad guesses left
Guess a letter: r
Good guess!
Your word: a--ar-
Bad choices: et
4 bad guesses left
Guess a letter: y
Good guess!
Your word: a--ary
Bad choices: et
4 bad guesses left
Guess a letter: i
Good guess!
Your word: a-iary
Bad choices: et
4 bad guesses left
Guess a letter: p
Good guess!
Your word: apiary
That's right!
Will you play another? <y/n> n
Bye
```

Замечания по программе

В программе 16.3 перегрузка операций отношения позволяет работать со строками так же, как с числовыми переменными:

```
while (guesses > 0 && attempt != target)
```

Такой подход проще, нежели использование, например, функции `strcmp()` со строками в стиле C. Программа применяет `find()` для проверки на повторное использование символа; если символ уже вводился, то он будет присутствовать либо в строке `badchars` (неудачные попытки), либо в строке `attempt` (удачные попытки):

```
if (badchars.find(letter) != string::npos
   || attempt.find(letter) != string::npos)
```

Переменная `npos` — это статический член класса `string`. Вспомните, что это — максимально возможное количество символов в объекте `string`. Поскольку нумерация символов в строке начинается с нуля, то это значение на единицу больше, чем максимально возможная позиция символа, и может использоваться для индикации неудачного поиска символа в строке.

Также в программе используется перегруженная операция += для добавления символов к строке:

```
badchars += letter;  // добавление символа к объекту string
```

Основная часть программы начинается с проверки на наличие введенного символа в загаданном слове:

```
int loc = target.find(letter);
```

Если переменная loc имеет допустимое значение, буква может быть помещена в соответствующую позицию строки ответа:

```
attempt[loc]=letter;
```

Однако данная буква символ может встречаться в загаданном слове несколько раз, поэтому программе приходится продолжать проверку. В программе используется необязательный второй аргумент функции find(), указывающий начальную позицию в строке, с которой нужно начинать поиск. Поскольку буква была найдена в позиции loc, следующий поиск должен начаться с позиции loc + 1. Цикл while будет повторять поиск до тех пор, пока не будет найдено больше ни одного вхождения символа. Обратите внимание, что find() сообщит об ошибке, если значение loc превысит длину строки:

```
// Проверка того, не появляется ли буква снова
loc = target.find(letter, loc + 1);
while (loc != string::npos)
{
   attempt[loc]=letter;
   loc = target.find(letter, loc + 1);
}
```

Другие возможности, предлагаемые классом string

Библиотека string поддерживает множество других возможностей для работы со строками. Среди этих возможностей, например, удаление части или всей строки, замена части или всей строки частью другой строки (или всей строкой), добавление и удаление данных из строки, сравнение частей строк и строк целиком, извлечение подстроки из строки, копирование одной строки в другую, обмен содержимым двух строк. Большинство этих функций перегружено и может работать как со строками в стиле C, так и с объектами string. Функции библиотеки string кратко описаны в приложении E, но несколько функций мы рассмотрим в этой главе.

Первым делом обратимся к возможности автоматического изменения размера строки. Что происходит, когда программа из листинга 16.3 дописывает букву в конец строки? Она не может просто увеличить размер строки, поскольку это может привести к использованию соседних областей памяти, которые уже заняты. Поэтому нужно выделить новый блок памяти и скопировать туда старое содержимое строки. Частое повторение такой процедуры приводило бы к снижению производительности, поэтому большинство реализаций C++ выделяют для строки блок памяти больший, чем фактическая строка, обеспечивая возможность ее увеличения. Когда со временем размер строки превышает размер этого блока, программа выделяет новый блок, вдвое больше текущего, обеспечивая дополнительное свободное место без постоянного изменения размера. Метод capacity() возвращает размер текущего блока, а метод reserve() позволяет запросить минимальный размер для блока. Пример использования этих методов приведен в листинге 16.4.

Листинг 16.4. str2.cpp

```
// str2.cpp -- использование методов capacity() и reserve()
#include <iostream>
#include <string>
int main()
{
   using namespace std;
   string empty;
   string small = "bit";
```

```
    string larger = "Elephants are a girl's best friend";
    cout << "Sizes:\n";
    cout << "\tempty: " << empty.size() << endl;
    cout << "\tsmall: " << small.size() << endl;
    cout << "\tlarger: " << larger.size() << endl;
    cout << "Capacities:\n";
    cout << "\tempty: " << empty.capacity() << endl;
    cout << "\tsmall: " << small.capacity() << endl;
    cout << "\tlarger: " << larger.capacity() << endl;
    empty.reserve(50);
    cout << "Capacity after empty.reserve(50): "
         << empty.capacity() << endl;
    return 0;
}
```

Вот как выглядит вывод программы из листинга 16.4 для одной из реализаций C++:

```
Sizes:
        empty :  0
        sma l l :  3
        larger: 34
Capacities:
        empty: 15
        small: 15
        larger: 47
Capacity after empty.reserve(50): 63
```

Обратите внимание, что в этой реализации для строки резервируется минимум 15 символов и, похоже, что стандартный шаг увеличения размера на единицу меньше значений, кратных 16. В других реализациях языка значения могут быть иными.

Но как поступить, если имеется объект `string`, а нужна строка в стиле C? Например, может требоваться открыть файл, имя которого хранится в объекте `string`:

```
string filename;
cout << "Enter file name: ";
cin >> filename;
ofstream fout;
```

Проблема в том, что метод `open()` требует в качестве аргумента строку в стиле C. Однако существует метод `c_str()`, который возвращает указатель на строку в стиле C с тем же содержимым, что и у вызывающего объекта `string`. Поэтому можно использовать следующий оператор:

```
fout.open(filename.c_str());
```

Разновидности строк

В этом разделе обработка класса `string` осуществляется так, как если бы он был построен на основе типа `char`. В действительности же, как было отмечено ранее, в основе библиотеки `string` лежит шаблонный класс:

```
template<class charT, class traits = char _traits<charT>,
        class Allocator = allocator<charT> >
basic_string {...};
```

Существуют четыре разновидности шаблона `basic_string`, каждая из которых имеет имя `typedef`:

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
typedef basic_string<char16_t> u16string;   // C++11
typedef basic_string<char32_t> u32string;   // C++11
```

Это позволяет использовать строки на основе типов wchar_t, char16_t и char32_t, а также строки на базе char. Более того, можно разработать свой класс на основе символьных типов и применять шаблон класса basic_string при условии, что новый класс удовлетворяет определенным требованиям. Класс traits описывает определенные аспекты выбранного символьного типа, например, способы сравнения значений. Существуют заранее определенные разновидности шаблона char_traits для типов char, wchar_t, char16_t и char32_t, и они служат значениями по умолчанию для класса traits. Класс Allocator предназначен для управления памятью. Доступны предварительно определенные разновидности шаблона allocator для различных типов символов, которые являются значениями по умолчанию. Они используют операции new и delete.

Классы шаблонов интеллектуальных указателей

Интеллектуальный указатель (smart pointer) — это объект класса, который действует подобно указателю, но обладает дополнительными возможностями. В этом разделе мы рассмотрим три шаблона интеллектуальных указателей, которые могут облегчить динамическое выделение памяти. Мы начнем с рассмотрения того, что может требоваться для решения этой задачи, и того, как ее можно выполнить. Рассмотрим следующую функцию:

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    str = ps;
    return;
}
```

Работа программы довольно понятна. При каждом вызове функции она выделяет память из кучи, однако никогда не освобождает память, что приводит к утечкам памяти. Решение проблемы известно — нужно лишь не забыть об освобождении памяти, для чего добавить следующую строку перед оператором return:

```
delete ps;
```

Однако решение, в основе которого лежит фраза "нужно лишь не забыть", редко является идеальным. Иногда об этом забывают. Или помнят, но случайно удаляют или комментируют строку эту строку кода. И даже в том случае, когда об этом помнят, могут возникать проблемы. Рассмотрим следующий вариант:

```
void remodel(std::string & str)
{
    std::string * ps = new std::string(str);
    ...
    if (weird_thing())
        throw exception();
    str = *ps;
    delete ps;
    return;
}
```

При возникновении исключения до операции delete дело не доходит, и это снова приводит к утечке памяти.

Это упущение можно исправить, как показано в главе 14, но было бы желательно располагать более надежным решением. Что же для этого нужно? Когда функция вроде remodel() завершает работу — успешно либо с генерацией исключения — локальные переменные удаляются из стека памяти, в результате чего память, занятая указателем ps, освобождается. Было бы неплохо, если бы память, на которую указывал ps, также освобождалась. Если бы указатель ps обладал деструктором, этот деструктор мог бы освобождать указанную область памяти по истечении срока существования ps. Таким образом, проблема состоит в том, что ps — это обычный указатель, а не объект класса, обладающий деструктором. Если бы он был объектом, деструктор освобождал бы память, на которую указывал объект, при удалении объекта.

Именно эта идея лежит в основе использования шаблонов auto_ptr, unique_ptr и shared_ptr. Шаблон auto_ptr является решением C++98. В C++11 шаблон auto_ptr объявлен устаревшим, а в качестве альтернативы предлагаются два других шаблона. Однако, несмотря на это, шаблон auto_ptr применялся в течение многих лет и может оказаться единственным вариантом, если компилятор не поддерживает два других шаблона.

Использование интеллектуальных указателей

Каждый из трех названных шаблонов интеллектуальных указателей (auto_ptr, unique_ptr и shared_ptr) определяет подобный указателю объект, которому присваивается адрес области памяти, полученный (прямо или косвенно) операцией new. Когда срок существования интеллектуального указателя истекает, его деструктор использует операция delete для освобождения памяти. Таким образом, присваивая адрес, возвращенный операцией new, одному из этих объектов, не нужно беспокоиться об освобождении памяти впоследствии; она будет освобождена автоматически при удалении объекта интеллектуального указателя. Различия в работе auto_ptr и обычного указателя показаны на рис. 16.2. В этой ситуации shared_ptr и unique_ptr ведут себя одинаково.

Чтобы создать один из этих объектов интеллектуальных указателей, потребуется включить заголовочный файл memory, который содержит определения шаблонов. Затем с помощью обычного синтаксиса шаблона создается необходимая разновидность указателя. Например, шаблон auto_ptr содержит следующий конструктор:

```
template<class X> class auto_ptr {
public:
     explicit auto_ptr(X* p =0) throw();
...};
```

(Нотация throw() означает, что конструктор не должен генерировать исключения. Как и auto_ptr, она считается устаревшей.) Таким образом, запрашивая объект auto_ptr типа X, мы получаем объект auto_ptr, который указывает на значение типа X:

```
auto_ptr<double> pd(new double);  // pd — объект auto_ptr, указывающий на
                                  // значение типа double (используется вместо double * pd)
auto_ptr<string> ps(new string);  // ps — объект auto_ptr, указывающий на
                                  // значение типа string (используется вместо string * ps)
```

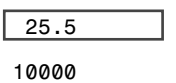
В этом примере new double — это указатель (возвращенный операцией new) на новый выделенный участок памяти. Он используется в качестве аргумента при вызове конструктора auto_ptr<double>, т.е. является фактическим аргументом, соответствующим формальному параметру p в прототипе класса.

```
void demo1()
{
     double * pd = new double; // #1
     *pd = 25.5;               // #2
     return;                   // #3
}
```

#1: Создается хранилище для pd и значения типа double, адрес сохраняется:

#2: Значение копируется в динамическую память:

#3: pd удаляется, значение остается в памяти:




```
void demo2()
{
     auto_ptr<double> ap(new double); // #1
     *ap = 25.5;                      // #2
     return;                          // #3
}
```

#1: Создается хранилище для pd и значения типа double, адрес сохраняется:

#2: Значение копируется в динамическую память:

#3: ap удаляется и деструктор объекта ap освобождает память.

Рис. 16.2. Обычный указатель и объект `auto_ptr`

Аналогично, `new string` также является фактическим аргументом конструктора. Остальные два интеллектуальные указатели используют тот же самый синтаксис:

```
unique_ptr<double> pdu(new double);     // pdu – объект unique_ptr,
                                        // указывающий на double
shared_ptr<string> pss(new string);     // pss – объект shared_ptr,
                                        // указывающий на string
```

Таким образом, чтобы преобразовать функцию `remodel()`, понадобится выполнить следующие три шага.

1. Включить заголовочный файл `memory`.
2. Заменить указатель на `string` объектом интеллектуального указателя, который указывает на `string`.
3. Удалить обращение к операции `delete`.

После этих изменений функция, использующая объект auto_ptr, приобретает следующий вид:

```
#include <memory>
void remodel(std::string & str)
{
   std::auto_ptr<std::string> ps (new std::string(str));
   ...
   if (weird_thing())
   throw exception();
   str = *ps;
   // delete ps; ЭТОТ ОПЕРАТОР БОЛЬШЕ НЕ НУЖЕН
   return;
}
```

Обратите внимание, что интеллектуальные указатели принадлежат пространству имен std. В листинге 16.5 приведена простая программа, в которой используются все три описанных выше интеллектуальных указателя. (Чтобы ее можно было выполнить, компилятор должен поддерживать классы shared_ptr и unique_ptr из C++11.) Каждый случай использования помещается внутрь блока, чтобы указатель удалялся, когда процесс выполнения программы покидает блок. Класс Report применяет многословные методы для сообщения о создании или уничтожении объекта.

Листинг 16.5. smrtptrs.cpp

```
// smrtptrs.cpp -- использование трех видов интеллектуальных указателей
// требуется поддержка shared_ptr и unique_ptr из C++11
#include <iostream>
#include <string>
#include <memory>

class Report
{
private:
   std::string str;
public:
   Report(const std::string s) : str(s)
      { std::cout << "Object created!\n"; }
   ~Report() { std::cout << "Object deleted!\n"; }
   void comment() const { std::cout << str << "\n"; }
};

int main()
{
   {
      std::auto_ptr<Report> ps (new Report("using auto_ptr"));
      ps->comment();  // использование операции -> для вызова функции-члена
   }
   {
      std::shared_ptr<Report> ps (new Report("using shared_ptr"));
      ps->comment();
   }
   {
      std::unique_ptr<Report> ps (new Report("using unique_ptr"));
      ps->comment();
   }
   return 0;
}
```

Ниже показан вывод этой программы:

```
Object created!
using auto_ptr
Object deleted!
Object created!
using shared_ptr
Object deleted!
Object created!
using unique_ptr
Object deleted!
```

Каждый из этих классов имеет конструктор `explicit`, который принимает указатель в качестве аргумента. Поэтому автоматическое преобразование типов из указателя в объект интеллектуального указателя не выполняется:

```
shared_ptr<double> pd;
double *p_reg = new double;
pd = p_reg;                              // недопустимо (неявное преобразование)
pd = shared_ptr<double>(p_reg);          // допустимо (явное преобразование)
shared_ptr<double> pshared = p_reg;      // недопустимо (неявное преобразование)
shared_ptr<double> pshared(p_reg);       // допустимо (явное преобразование)
```

Классы интеллектуальных указателей определены таким образом, что в большинстве случаев объект интеллектуального указателя работает подобно обычному указателю. Например, если `ps` — объект интеллектуального указателя, его можно разыменовывать (`*ps`), использовать для получения доступа к членам структуры (`ps->puffIndex`) и присваивать регулярному указателю, который указывает на тот же тип. Один объект интеллектуального указателя можно также присвоить другому того же типа, но при этом возникают проблемы, которые рассмотрены в следующем разделе. Но вначале давайте остановимся на ситуации, которой следует избегать при использовании всех трех названных интеллектуальных указателей:

```
string vacation("I wandered lonely as a cloud.");
shared_ptr<string> pvac(&vacation); // ТАК ДЕЛАТЬ НЕЛЬЗЯ!
```

При удалении объекта `pvac` программа применила бы операцию `delete` к памяти не из кучи, что совершенно неприемлемо.

Если код в листинге 16.5 представляет собой венец программных устремлений, любой из этих трех интеллектуальных указателей может быть использован с равным успехом. Однако их возможности не ограничиваются уже описанными.

Соображения по поводу интеллектуальных указателей

Зачем нужны три интеллектуальных указателя? (В действительности их четыре, но указатель `weak_ptr` обсуждаться не будет.) И почему отказались от указателя `auto_ptr`?

Начнем с рассмотрения следующего присваивания:

```
auto_ptr<string> ps (new string("I reigned lonely as a cloud."));
auto_ptr<string> vocation;
vocation = ps;
```

Каким должен быть результат этого оператора присваивания? Если бы `ps` и `vocation` были обычными указателями, результатом стали бы два указателя на один и тот же объект `string`. В данном случае это недопустимо, поскольку программа вела бы себя непредсказуемо, пытаясь удалить один объект дважды — при удалении `ps` и при удалении `vocation`.

Существуют следующие способы предотвращения этой проблемы.

- Определить операцию присваивания так, чтобы она создавала точную копию объекта. Тогда два указателя будут указывать на два разных объекта, один из которых является копией второго.
- Использовать концепцию *владения*, когда определенным объектом может владеть только один интеллектуальный указатель. Деструктор будет удалять объект только тогда, когда интеллектуальный указатель владеет объектом. Затем можно использовать операцию присваивания, которая будет передавать право владения объектом. Эта стратегия применяется указателями `auto_ptr` и `unique_ptr`, но `unique_ptr` накладывает несколько больше ограничений.
- Создать еще более интеллектуальный указатель, который будет отслеживать, сколько интеллектуальных указателей ссылается на определенный объект. Эта стратегия называется *подсчетом ссылок*. Например, присваивание могло бы увеличивать значение счетчика на единицу, а удаление указателя — уменьшать его. Тогда операция `delete` вызвалась бы только при удалении последнего указателя. Эта стратегия применяется для указателя `shared_ptr`.

Те же самые стратегии применимы и к конструкторам копирования.

Каждый подход находит свое применение. В листинг 16.6 представлен пример, в котором указатель `auto_ptr` не особенно хорошо подходит.

Листинг 16.6. `fowl.cpp`

```
// fowl.cpp -- auto_ptr - неудачный выбор
#include <iostream>
#include <string>
#include <memory>

int main()
{
   using namespace std;
   auto_ptr<string> films[5] =
   {
      auto_ptr<string> (new string("Fowl Balls")),
      auto_ptr<string> (new string("Duck Walks")),
      auto_ptr<string> (new string("Chicken Runs")),
      auto_ptr<string> (new string("Turkey Errors")),
      auto_ptr<string> (new string("Goose Eggs"))
   };
   auto_ptr<string> pwin;
   pwin = films[2];   // films[2] утрачивает права владения

   cout << "The nominees for best avian baseball film are\n";
   for (int i = 0; i < 5; i++)
   cout << *films[i] << endl;
   cout << "The winner is " << *pwin << "!\n";
   cin.get();
   return 0;
}
```

Ниже показан пример вывода этой программы:

```
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Segmentation fault (core dumped)
```

Сообщение "Segmentation 'fault' (core dumped)" (Ошибка сегментации (дамп ядра записан)) должно служить напоминанием о том, что неправильное использование объекта auto_ptr может привести к возникновению проблем. (Поведение такого кода неопределенно, поэтому можно столкнуться с различным поведением в зависимости от конкретной системы.) В данном случае проблема заключается в том, что следующий оператор передает права владения от films[2] к pwin:

```
pwin = films[2];  // films[2] утрачивает права владения
```

Это ведет к тому, что элемент массива films[2] перестает ссылаться на строку. После того, как auto_ptr передает права владения объектом, он больше не предоставляет доступ к этому объекту. Когда программа приступает к выводу строки, указанной элементом films[2], она обнаруживает нулевой указатель, что, несомненно, оказывается неприятным сюрпризом.

Предположим, что мы возвратились к листингу 16.6, но вместо auto_ptr использовали shared_ptr. (Компилятор должен поддерживать класс shared_ptr из C++11.) В этом случае программа выполняется успешно и дает следующий вывод:

```
The nominees for best avian baseball film are
Fowl Balls
Duck Walks
Chicken Runs
Turkey Errors
Goose Eggs
The winner is Chicken Runs!
```

Различие состоит в следующей части программы:

```
shared_ptr<string> pwin;
pwin = films[2];
```

На этот раз и pwin, и films[2] указывают на один и тот же объект, а значение счетчика ссылок увеличивается с 1 до 2. В конце программы объект pwin, который был объявлен последним, оказывается первым, чей деструктор будет вызван. Деструктор уменьшает значение счетчика ссылок до 1. Затем элементы массива shared_ptrs освобождаются. Деструктор films[2] уменьшает значение счетчика до 0 и освобождает ранее выделенную память.

Это же относится к shared_ptr. Программа из листинга 16.6 выполняется успешно. При использовании auto_ptr происходит ошибка времени выполнения. А что происходит в случае применения unique_ptr? Как и auto-ptr, unique_ptr использует модель владения. Однако вместо сбоя версия с unique_ptr генерирует ошибку во время компиляции следующей строки кода:

```
pwin = films[2];
```

Очевидно, что пора подробнее рассмотреть различия между этими двумя последними типами.

Почему unique_ptr предпочтительней auto_ptr

Рассмотрим следующие операторы:

```
auto_ptr<string> p1(new string("auto");     //#1
auto_ptr<string> p2;                        //#2
p2 = p1;                                    //#3
```

Когда в операторе #3 указатель p2 получает права владения объектом string, указатель p1 лишается этих прав. Это хорошо, поскольку препятствует попыткам дест-

рукторов обоих объектов p1 и p2 удалить один и тот же объект. Но это же и плохо, если впоследствии программа пытается использовать p1, т.к. p1 больше не указывает на соответствующие данные. Теперь рассмотрим эквивалентную программу, в которой применяется unique_ptr:

```
unique_ptr<string> p3(new string("auto");   //#4
unique_ptr<string> p4;                      //#5
p4 = p3;                                    //#6
```

В этом случае компилятор не разрешает выполнение оператора #6, и мы избегаем проблемы, связанной с тем, что p3 не указывает на соответствующие данные. Таким образом, unique_ptr безопаснее auto_ptr (поскольку вызывает ошибку времени компиляции, а не приводит к сбою программы).

Но в некоторых случаях присваивание одного интеллектуального указателя другому не устраняет потенциальные проблемы. Предположим, имеется следующее определение функции:

```
unique_ptr<string> demo(const char * s)
{
   unique_ptr<string> temp(new string(s));
   return temp;
}
```

Также представим, что используется такой код:

```
unique_ptr<string> ps;
ps = demo("Uniquely special");
```

В этом примере функция demo() возвращает временный объект unique_ptr, а затем ps принимаст права владения объектом, первоначально принадлсжавшего возвращенному объекту unique_ptr. После этого возвращенный объект unique_ptr уничтожается. Это нормально, поскольку теперь объект ps владеет объектом string. Но при этом имеет место еще один положительный нюанс. Поскольку временный объект unique_ptr, возвращенный функцией demo(), вскоре уничтожается, отсутствует какая-либо возможность его неправильного использования для доступа к недопустимым данным. Иначе говоря, в данном случае нет никакой причины запрещать присваивание. И, как не удивительно, компилятор его разрешает!

Короче говоря, если программа пытается присвоить один объект unique_ptr другому, компилятор не препятствует этому, если исходный объект является временным значением, и запрещает это, если исходный объект существует некоторое время:

```
using namespace std;
unique_ptr< string> pu1(new string "Hi ho!");
unique_ptr< string> pu2;
pu2 = pu1;                                  // #1 не разрешено
unique_ptr<string> pu3;
pu3 = unique_ptr<string>(new string "Yo!"); // #2 разрешено
```

Оператор присваивания #1 оставил бы висячий объект unique_ptr (объект pu1) — возможный источник ошибки. Оператор присваивания #2 не оставляет за собой никакого объекта unique_ptr, поскольку он вызывает конструктор unique_ptr, создающий временный объект, уничтожаемый при передаче прав владения объекту pu3. Это избирательное поведение — одна из причин того, что шаблон unique_ptr предпочтительнее auto_ptr, который допускал бы обе формы присваивания. По этой же причине использование объектов auto_ptr в контейнерных объектах запрещастся (рекомендацией, но не компилятором), в то время как применение объектов unique_ptr

разрешено. Если алгоритм контейнера пытается выполнить с содержимым контейнера unique_ptr что-либо аналогичное строкам присваивания #1, это ведет к ошибке времени компиляции. Если же алгоритм пытается выполнить что-то вроде присваивания #2, то все проходит нормально, и выполнение программы продолжается. При использовании объектов auto_ptr действия, подобные присваиванию #1, могли бы вести к непрогнозируемому поведению и непонятным сбоям программы.

Конечно, в определенных ситуациях действительно может требоваться выполнение действий, подобных присваиванию #1. Присваивание небезопасно только при неинтеллектуальном использовании отброшенного интеллектуального указателя, например, при его разыменовании. Но указатель можно безопасно использовать, присвоив ему новое значение. В C++ имеется стандартная библиотечная функция std::move(), которая позволяет присваивать один объект unique_ptr другому.

Ниже приведен пример применения ранее определенной функции, которая возвращает объект unique_ptr<string>:

```
using namespace std;
unique_ptr<string> ps1, ps2;
ps1 = demo("Uniquely special");
ps2 = move(ps1);                          // делает возможным присваивание
ps1 = demo(" and more");
cout << *ps2 << *ps1 << endl;
```

Может возникать вопрос, каким образом unique_ptr, в отличие от auto_ptr, способен различить безопасное и потенциально опасное использование. Ответ заключается в том, что он использует дополнения конструкторов переноса и ссылок rvalue из C++11, которые описаны в главе 18.

Кроме того, unique_ptr обладает еще одним преимуществом по сравнению с auto_ptr. Он имеет вариант, который можно использовать с массивами. Вспомните, что операция delete применяется только в паре с new, а delete [] — только в паре с new []. Шаблон auto_ptr использует операцию delete, а не delete [], поэтому может применяться только с new, но не с new []. Однако unique_ptr имеет версию для пары new[] и delete[]:

```
std::unique_ptr< double[]>pda(new double(5)); // будет использовать delete []
```

> Объект auto_ptr или shared_ptr должен использоваться только для памяти, выделенной операцией new. Память, выделенная с помощью new[], не подходит. Нельзя применять auto_ptr, shared_ptr или unique_ptr для памяти, выделенной посредством операции new либо, в случае unique_ptr, с помощью new или new[].

Выбор интеллектуального указателя

Какой тип указателя следует использовать? Если в программе требуется более одного указателя на объект, необходимо выбрать shared_ptr. Например, может существовать массив указателей, а несколько вспомогательных указателей применяться для идентификации определенных элементов, таких как максимальный и минимальный. Или же возможно наличие двух видов объектов, которые содержат указатели на один и тот же третий объект. Либо можно располагать контейнером указателей из STL. Многие алгоритмы STL включают в себя операции копирования или присваивания, которые будут работать с объектом shared_ptr, но не с unique_ptr (компилятор будет выводить предупреждение) либо auto_ptr (это будет приводить к непредсказуемому поведению). Если компилятор не разрешает применять shared_ptr, можно получить версию из библиотеки BOOST.

Если программа не нуждается в нескольких указателях на один и тот же объект, unique_ptr работает вполне успешно. Это хороший вариант для возвращаемого типа функции, которая возвращает указатель на память, выделенную операцией new. В результате права владения передаются объекту unique_ptr, которому присвоено возвращаемое значение, и интеллектуальный указатель принимает на себя ответственность за вызов операции delete. Объекты unique_ptr можно сохранять в контейнере STL, если только не требуется вызывать методы или алгоритмы, такие как sort(), которые копируют или присваивают один объект unique_ptr другому. Например, при условии наличия соответствующих операторов include и using, в программе можно было бы использовать фрагменты кода вроде показанных ниже:

```
unique_ptr<int> make_int(int n)
{
   return unique_ptr<int>(new int(n));
}
void show(unique_ptr<int> & pi)          // передача по ссылке
{
   cout << *a << ' ';
}
int main()
{
...
  vector<unique_ptr<int> > vp(size);
  for (int i = 0; i < vp.size(); i++)
     vp[i] = make_int(rand() % 1000);    // копирование временного unique_ptr
  vp.push_back(make_int(rand() % 1000)) // номально, поскольку аргумент
                                        // является временным
  for_each(vp.begin(), vp.end(), show); // использование for_each()
...
}
```

Вызов функции push_back() работает, поскольку он передает временный объект unique_ptr, который должен быть присвоен объекту unique_ptr в vp. Также обратите внимание, что оператор for_each() приводил бы к ошибке, если бы функция show() передавала объект по значению, а не посредством ссылки, поскольку в этом случае было бы необходимо инициализировать pi значением unique_ptr из vp, не являющимся временным, что недопустимо. Как уже упоминалось, компилятор будет перехватывать попытки неправильного использования объекта unique_ptr.

Объект unique_ptr можно присваивать объекту shared_ptr при соблюдении тех же условий, при которых один объект unique_ptr допускается присваивать другому — источником должно быть rvalue. Как и ранее, в следующем коде make_int() представляет собой функцию, возвращаемым типом которой является unique_ptr<int>:

```
unique_ptr<int> pup(make_int(rand() % 1000); // нормально
shared_ptr<int> spp(pup);                    // недопустимо, pup — это lvalue
shared_ptr<int> spr(make_int(rand() % 1000); // нормально
```

Шаблон shared_ptr содержит явный конструктор преобразования rvalue типа unique_ptr в shared_ptr. При этом shared_ptr принимает права владения объектом, первоначально принадлежавшего unique_ptr.

Объект auto_ptr можно было бы использовать в тех же ситуациях, что и unique_ptr, но последний предпочтительнее. Если компилятор не поддерживает unique_ptr, можно подумать о применении класса scoped_ptr из библиотеки BOOST, который предлагает аналогичные возможности.

Стандартная библиотека шаблонов (STL)

Библиотека STL содержит набор шаблонов, представляющих контейнеры, итераторы, объекты функций и алгоритмы. Контейнер – это структура данных, похожая на массив, которая может хранить несколько значений. Контейнеры STL однородны по структуре и хранят только однотипные значения. Алгоритмы используются для решения определенных задач, например, сортировки массива или поиска конкретного значения в списке. Итераторы – это объекты, позволяющие перемещаться внутри контейнера подобно тому, как указатели позволяют перемещаться по массиву; они являются обобщениями указателей. Объекты функций – это объекты, которые ведут себя подобно функциям; они могут быть объектами класса или указателями на функции (в том числе именами функций, поскольку имя функции действует как указатель). Библиотека STL позволяет создавать различные контейнеры, включая массивы, очереди и списки, и осуществлять множество операций, например, поиск, сортировку и тасование в случайном порядке.

Алекс Степанов (Alex Stepanov) и Менг Ли (Meng Lee) разработали STL в лаборатории Hewlett-Packard в 1994 г. Комитет по стандарту ISO/ANSI C++ проголосовал за внедрение библиотеки в стандарт C++. STL не является примером объектно-ориентированного программирования. В ней используется другая идеология программирования – *обобщенное программирование*. Поэтому библиотека STL интересна как с точки зрения предлагаемых возможностей, так и с точки зрения применяемого в ней подхода.

Информация по STL слишком обширна для одной главы, поэтому здесь будут представлены только некоторые показательные примеры ее использования, позволяющие ощутить дух подхода обобщенного программирования. Мы начнем с рассмотрения нескольких конкретных примеров. После этого, когда будет достигнуто достаточное понимание работы контейнеров, итераторов и алгоритмов, мы рассмотрим философию построения архитектуры библиотеки и приведем обзор библиотеки STL в целом. Краткое описание различных методов и функций STL приведено в приложении Ж.

Класс шаблона `vector`

В главе 4 мы вскользь коснулись класса `vector`, а теперь рассмотрим его подробней. В вычислительной технике термин *вектор* соответствует массиву, а не математическим векторам, которые обсуждались в главе 11. (С точки зрения математики N-мерный математический вектор может быть представлен набором из N компонентов, и в этом смысле математический вектор подобен на N-мерному массиву. Однако математический вектор обладает дополнительными свойствами, такими как скалярное и векторное произведение, которые для компьютерного вектора не обязательны.) Компьютерный вектор – это набор однотипных значений, к которым можно обращаться в произвольном порядке. То есть, например, с помощью индекса можно получить непосредственный доступ к десятому элементу вектора без необходимости считывания девяти предыдущих элементов. Итак, класс `vector` должен обладать возможностями, аналогичными предоставляемым классами `valarray` и `ArrayTP` (см. главу 14), а также классом `array` (см. главу 4). Это значит, что можно создать объект `vector`, присвоить один объект `vector` другому и применять операцию `[]` для доступа к отдельным элементам объекта `vector`. Чтобы сделать этот класс обобщенным, его нужно реализовать в виде класса шаблона. Именно это и делает библиотека STL, определяя шаблон `vector` в заголовочном файле `vector` (ранее известный как `vector.h`).

Для создания объекта шаблона vector используется обычная нотация *<тип>*, с помощью которой указывается необходимый тип. Кроме того, шаблон vector использует динамическое выделение памяти, и для указания количества элементов вектора можно применять инициализирующий аргумент:

```
#include vector
using namespace std;
vector<int> ratings(5);       // вектор из 5 значений типа int
int n;
cin >> n;
vector<double> scores(n);     // вектор из n значений типа double
```

После создания объекта vector перегрузка операции [] позволяет обращаться к элементам вектора, используя обычную нотацию массивов:

```
ratings[0] = 9;
for (int i = 0; i < n; i++)
   cout << scores[i] << endl;
```

Еще раз о распределителях

Подобно классу string, различные шаблоны контейнеров STL принимают необязательный аргумент, который указывает, какой объект-распределитель будет использоваться для управления памятью. Например, шаблон vector начинается с таких строк:

```
template <class T, class Allocator = allocator<T> >
     class vector {...
```

Если опустить значение этого аргумента, шаблон контейнера по умолчанию будет применять класс allocator<T>. Этот класс использует операции new и delete.

Применение класса vector в достаточно непритязательном приложении продемонстрировано в листинге 16.7. Эта программа создает два объекта vector, один из которых содержит элементы типа int, а второй — string. В каждом объекте находится по 5 элементов.

Листинг 16.7. vect1.cpp

```
// vect1.cpp -- пример работы с шаблоном vector
#include <iostream>
#include <string>
#include <vector>

const int NUM = 5;
int main()
{
   using std::vector;
   using std::string;
   using std::cin;
   using std::cout;
   using std::endl;

   vector<int> ratings(NUM);
   vector<string> titles(NUM);
   cout << "You will do exactly as told. You will enter\n"
       << NUM << " book titles and your ratings (0-10).\n";
              // запрос книг и их рейтингов
   int i;
   for (i = 0; i < NUM; i++)
   {
      cout << "Enter title #" << i + 1 << ": ";  // ввод названия книги
```

```
        getline(cin,titles[i]);
        cout << "Enter your rating (0-10): ";     // ввод рейтинга книги
        cin >> ratings[i];
        cin.get();
    }
    cout << "Thank you. You entered the following:\n"
         << "Rating\tBook\n";                     // вывод списка книг с рейтингами
    for (i = 0; i < NUM; i++)
    {
        cout << ratings[i] << "\t" << titles[i] << endl;
    }
    return 0;
}
```

Ниже приведен пример запуска программы из листинга 16.7:

```
You will do exactly as told. You will enter
5 book titles and your ratings (0-10).
Enter title #1: The Cat Who Knew C++
Enter your rating (0-10): 6
Enter title #2: Felonious Felines
Enter your rating (0-10): 4
Enter title #3: Warlords of Wonk
Enter your rating (0-10): 3
Enter title #4: Don't Touch That Metaphor
Enter your rating (0-10): 5
Enter title #5: Panic Oriented Programming
Enter your rating (0-10): 8
Thank you. You entered the following:
Rating  Book
6       The Cat Who Knew C++
4       Felonious Felines
3       Warlords of Wonk
5       Don't Touch That Metaphor
8       Panic Oriented Programming
```

Все, что делает эта программа — использует шаблон `vector` в качестве удобного средства создания динамического массива. В следующем разделе будет показан пример применения других методов этого класса.

Что еще можно делать с помощью векторов

Что же еще, помимо выделения памяти, позволяет делать шаблон `vector`? Все контейнеры STL предоставляют набор базовых методов. К ним относятся: `size()`, который возвращает количество элементов в контейнере; `swap()`, обменивающий содержимое двух контейнеров; `begin()`, возвращающий итератор, который ссылается на первый элемент в контейнере; `end()`, возвращающий итератор, который представляет область памяти, следующую за последним элементом контейнера.

Что собой представляет итератор? Это обобщение указателя. В действительности он может быть указателем. Или же он может быть объектом, для которого определены операции над указателями, такие как разыменование (например, `operator*()`) и инкремент (например, `operator++()`). Как станет видно в дальнейших примерах, обобщение указателей позволяет STL предоставлять однотипный интерфейс для множества классов-контейнеров, включая те, для которых обычные указатели не работают. В каждом классе-контейнере определяется соответствующий итератор. Типом

этого итератора будет typedef по имени iterator с областью видимости класса. Например, для объявления итератора для класса vector типа double применяется следующий синтаксис:

```
vector<double>::iterator pd; // pd – это итератор
```

Предположим, что scores – это объект vector<double>:

```
vector<double> scores;
```

Теперь итератор pd можно применять в коде, как показано ниже:

```
pd = scores.begin(); // обеспечение того, чтобы pd указывал на первый элемент
*pd = 22.3;          // разыменование pd и присваивание значения первому элементу
++pd;                // обеспечение того, чтобы pd указывал на следующий элемент
```

Как видите, итератор ведет себя подобно указателю. Кстати говоря, автоматическое выведение типа C++11 может быть полезно еще в одной ситуации. Вместо оператора:

```
vector<double>::iterator pd = scores.begin();
```

можно использовать следующий оператор:

```
auto pd = scores.begin();  // автоматическое выведение типа C++11
```

Но что подразумевается в примере под областью памяти, *следующей за последним элементом?* Это итератор, который указывает на элемент, следующий за последним элементом в контейнере. Идея применения этого итератора сходна с идеей использования нулевого символа, расположенного непосредственно за последним символом в строке стиля C, за исключением того, что нулевой символ – это значение элемента, а следующий за последним элемент – это указатель (или итератор) элемента. Функция-член end() определяет позицию, следующую за последним элементом контейнера. Если установить итератор на первый элемент контейнера и увеличивать его, то, в конце концов, будет достигнут элемент, следующий за последним – т.е. все содержимое контейнера было пройдено. Таким образом, если scores и pd определены как в предыдущем примере, то все содержимое контейнера можно отобразить с помощью следующего кода:

```
for (pd = scores.begin(); pd != scores.end(); pd++)
   cout << *pd << endl;;
```

Описанные выше методы имеются у всех контейнеров. В шаблоне vector есть также некоторые методы, которые присутствуют не во всех контейнерах STL. Один из полезных методов – push_back() – добавляет элемент в конец объекта vector. При выполнении этой операции осуществляется дополнительное выделение памяти, и размер вектора увеличивается, чтобы в него поместились добавляемые элементы. Это означает, что можно использовать код, подобный следующему:

```
vector<double> scores;  // создание пустого вектора
double temp;
while (cin >> temp && temp >= 0)
   scores.push_back(temp);
cout << "You entered " << scores.size() << " scores.\n";
```

На каждом проходе цикла в объект scores добавляется один элемент. Во время создания или запуска программы не нужно заботиться о количестве элементов. До тех пор, пока у программы есть доступ к необходимому объему памяти, размер scores будет при необходимости увеличиваться.

Метод `erase()` удаляет данный диапазон элементов вектора. В качестве аргументов он принимает два итератора, которые определяют границы диапазона удаляемых элементов. Важно понимать, как STL определяет диапазон, заданный итераторами. Первый итератор указывает на начало диапазона, второй — на элемент, следующий за концом диапазона. Например, следующий код удаляет первый и второй элементы — те, на которые ссылаются `begin()` и `begin() + 1`:

```
scores.erase(scores.begin(), scores.begin() + 2);
```

(Поскольку `vector` предоставляет непосредственный доступ к любому элементу, такие операции, как `begin() + 2`, определены для итераторов класса `vector`.) В литературе по STL используется также запись вида [p1, p2) (где p1 и p2 — итераторы), описывающая диапазон, начинающийся с p1 и заканчивающийся, но не включающий, p2. Таким образом, диапазон [`begin()`, `end()`) охватывает все содержимое коллекции (рис. 16.3). Кроме того, запись [p1, p1) определяет пустой диапазон. (Нотация `[)` не является частью языка C++, поэтому в коде она не используется и присутствует только в документации.)

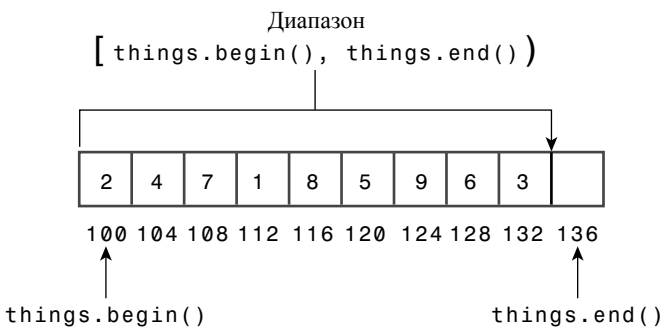


Рис. 16.3. *Концепция диапазона в STL*

На заметку!

Диапазон [`it1`, `it2`) указан двумя итераторами `it1` и `it2`; он начинается с `it1` и продолжается до `it2`, но не включает его.

Метод `insert()` дополняет `erase()`. В качестве аргументов он принимает три итератора. Первый указывает позицию, после которой будут добавляться новые элементы. Второй и третий итераторы описывают добавляемый диапазон. Обычно этот диапазон является частью другого объекта контейнера. Например, следующий код вставляет все элементы вектора `new_v`, за исключением первого, после первого элемента вектора `old_v`:

```
vector<int> old_v;
vector<int> new_v;
...
old_v.insert(old_v.begin(), new_v.begin() + 1, new_v.end());
```

Кстати, здесь может пригодиться метод `end()`, поскольку он облегчает добавление элементов в конец вектора. В этом коде новые данные добавляются после позиции `old.end()`, т.е. *после* последнего элемента вектора:

```
old_v.insert(old_v.end(), new_v.begin() + 1, new_v.end());
```

Код в листинге 16.8 иллюстрирует применение методов `size()`, `begin()`, `end()`, `push_back()`, `erase()` и `insert()`. Для упрощения работы с данными компоненты `rating` и `title` объединены в одну структуру `Review`, а функции `FillReview()` и `ShowReview()` предоставляют возможности ввода и вывода для объектов `Review`.

Листинг 16.8. vect2.cpp

```
// vect2.cpp -- методы и итераторы
#include <iostream>
#include <string>
#include <vector>
struct Review {
   std::string title;
   int rating;
};
bool FillReview(Review & rr);
void ShowReview(const Review & rr);
int main()
{
   using std::cout;
   using std::vector;
   vector<Review> books;
   Review temp;
   while (FillReview(temp))
      books.push_back(temp);
   int num = books.size();
   if (num > 0)
   {
      cout << "Thank you. You entered the following:\n"
           << "Rating\tBook\n";
      for (int i = 0; i < num; i++)
         ShowReview(books[i]);
      cout << "Reprising:\n"
           << "Rating\tBook\n";
      vector<Review>::iterator pr;
      for (pr = books.begin(); pr != books.end(); pr++)
         ShowReview(*pr);
      vector <Review> oldlist(books);  // использование конструктора копирования
      if (num > 3)
      {
         // Удаление двух элементов
         books.erase(books.begin() + 1, books.begin() + 3);
         cout << "After erasure:\n";
         for (pr = books.begin(); pr != books.end(); pr++)
            ShowReview(*pr);
         // Вставка одного элемента
         books.insert(books.begin(), oldlist.begin() + 1, oldlist.begin() + 2);
         cout << "After insertion:\n";
         for (pr = books.begin(); pr != books.end(); pr++)
            ShowReview(*pr);
      }
      books.swap(oldlist);
      cout << "Swapping oldlist with books:\n";
      for (pr = books.begin(); pr != books.end(); pr++)
         ShowReview(*pr);
   }
   else
      cout << "Nothing entered, nothing gained.\n";
   return 0;
}
```

```
bool FillReview(Review & rr)
{
   std::cout << "Enter book title (quit to quit): ";
   std::getline(std::cin,rr.title);
   if (rr.title == "quit")
      return false;
   std::cout << "Enter book rating: ";
   std::cin >> rr.rating;
   if (!std::cin)
      return false;
   // Избавиться от остальной части строки ввода
   while (std::cin.get() != '\n')
      continue;
   return true;
}
void ShowReview(const Review & rr)
{
   std::cout << rr.rating << "\t" << rr.title << std::endl;
}
```

Ниже приведен пример запуска программы из листинга 16.8:

```
Enter book title (quit to quit): The Cat Who Knew Vectors
Enter book rating: 5
Enter book title (quit to quit): Candid Canines
Enter book rating: 7
Enter book title (quit to quit): Warriors of Wonk
Enter book rating: 4
Enter book title (quit to quit): Quantum Manners
Enter book rating: 8
Enter book title (quit to quit): quit
Thank you. You entered the following:
Rating Book
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
Reprising:
Rating Book
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
After erasure:
5       The Cat Who Knew Vectors
8       Quantum Manners
After insertion:
7       Candid Canines
5       The Cat Who Knew Vectors
8       Quantum Manners
Swapping oldlist with books:
5       The Cat Who Knew Vectors
7       Candid Canines
4       Warriors of Wonk
8       Quantum Manners
```

Дополнительные возможности векторов

Существует множество задач, которые часто выполняются с массивами, такие как поиск, сортировка, тасование и т.д. Содержит ли класс шаблона вектора методы для выполнения этих часто выполняемых операций? Нет! В STL применяется более широкий подход, заключающийся в определении *автономных* (не являющихся членами класса) функций для выполнения этих операций. Таким образом, вместо того чтобы определять отдельную функцию-член `find()` для каждого класса контейнера, в библиотеке определяется одна автономная функция `find()`, которая может использоваться для всех классов контейнеров. Такой подход позволяет значительно уменьшить дублирование кода. Например, предположим, что имеется 8 контейнеров и 10 операций над ними. Если бы в каждом классе определялась своя функция-член для каждой операции, то потребовалось бы 8×10, или 80, отдельных определений функций-членов. Но при использовании подхода, применяемого в STL, потребуется всего лишь 10 отдельных определений автономных функций. И при соблюдении соответствующих рекомендаций 10 существующих автономных функций можно было бы использовать для выполнения операций поиска, сортировки и т.д. также в определении нового класса контейнера.

Вместе с тем, в ряде случаев STL определяет функцию-член даже при наличии автономной функции для решения той же самой задачи. Причина этого в том, что специфичные для класса алгоритмы выполнения некоторых действий эффективнее более общих алгоритмов. Поэтому функция `swap()` класса `vector` будет эффективнее автономной функции `swap()`. С другой стороны, автономная версия позволит обменивать содержимое двух контейнеров различного типа.

Давайте более подробно рассмотрим три типичных функции STL: `for_each()`, `random_shuffle()` и `sort()`. Функция `for_each()` работает с любым классом-контейнером. Она принимает три аргумента. Первые два аргумента – это итераторы, определяющие диапазон, а третий аргумент – указатель на функцию. (В более общем смысле последний аргумент представляет собой объект функции (функтор). Функторы рассматриваются далее в этой главе.) Затем функция `for_each()` применяет указанную в аргументе функцию ко всем элементам контейнера в указанном диапазоне. Функция, указанная в аргументе, не должна изменять значение элементов контейнера. Функцию `for_each()` можно использовать вместо цикла `for`. Например, код

```
vector<Review>::iterator pr;
for (pr = books.begin(); pr != books.end(); pr++)
   ShowReview(*pr);
```

можно заменить следующим кодом:

```
for_each(books.begin(), books.end(), ShowReview);
```

Это позволяет избежать явного использования переменных итераторов.

Функция `random_shuffle()` принимает в качестве аргументов два итератора, которые указывают границы диапазона, и тасует элементы в этом диапазоне случайным образом. Например, следующий оператор изменяет случайным образом порядок следования всех элементов вектора `books`:

```
random_shuffle(books.begin(), books.end());
```

В отличие от функции `for_each()`, которая работает с любым классом-контейнером, `random_shuffle()` требует, чтобы класс контейнера разрешал доступ к своим элементам в произвольном порядке. Класс `vector` удовлетворяет этому требованию.

Функция sort() также требует, чтобы контейнер поддерживал произвольный доступ. Существуют две версии этой функции. Первая использует два итератора, определяющих границы диапазона, и сортирует элементы этого диапазона с помощью операции <, определенной для типа элемента, который хранится в контейнере. Например, следующий код сортирует содержимое coolstuff по возрастанию с применением встроенной операции < для сравнения значений:

```
vector<int> coolstuff;
...
sort(coolstuff.begin(), coolstuff.end());
```

Если элементами контейнера являются объекты, типы которых определены пользователем, то для этого типа объектов должна быть определена функция operator<(), в противном случае функция sort() работать не будет. Например, вектор, содержащий объекты Review, можно было бы сортировать либо с помощью функции-члена класса Review, либо с помощью автономной функции operator<(). Поскольку Review — это структура, ее члены открыты, и в этом случае можно применять автономную функцию:

```
bool operator<(const Review & r1, const Review & r2)
{
    if (r1.title < r2.title)
        return true;
    else if (r1.title == r2.title && r1.rating < r2.rating)
        return true;
    else
        return false;
}
```

Используя подобную функцию, можно отсортировать вектор объектов Review (такой как books):

```
sort(books.begin(), books.end());
```

Эта версия функции operator<() сортирует члены title в лексикографическом порядке. Если у двух объектов поля title совпадают, объекты сортируются по полю rating. Но предположим, что требуется выполнить сортировку в убывающем порядке или по рейтингам rating, а не по заглавиям title. В этом случае можно использовать вторую форму функции sort(). Она принимает три аргумента. Первые два, как и в предыдущем случае, являются итераторами, определяющими диапазон. Третий аргумент — указатель на функцию (точнее — функтор), которая будет использоваться вместо operator<() для выполнения сравнения. Функция должна возвращать значение, которое можно преобразовать в тип bool, причем значение false означает, что аргументы функции расположены в неправильном порядке. Вот пример такой функции:

```
bool WorseThan(const Review & r1, const Review & r2)
{
    if (r1.rating < r2.rating)
        return true;
    else
        return false;
}
```

Располагая этой функцией, можно написать следующий оператор для сортировки вектора books, состоящего из объектов Review, по возрастанию рейтинга:

```
sort(books.begin(), books.end(), WorseThan);
```

Обратите внимание, что функция `WorseThan()` сортирует объекты `Review` менее точно, чем `operator<()`. Если член `title` объектов совпадает, функция `operator<()` осуществляет сортировку по полю `rating`. Но если и эти два поля объектов совпадают, функция `WorseThan()` считает их эквивалентными. Первый вид упорядочения называется *полным упорядочением*, а второй — *строгим квазиупорядочением*. При полном упорядочении, если оба выражения $a < b$ и $b < a$ ложны, то a должно быть идентично b. При строгом квазиупорядочении это не так. Объекты могут быть полностью идентичными, а могут совпадать только по одному критерию, такому как поле `rating` в примере с функцией `WorseThan()`. Поэтому при строгом квазиупорядочении лучше говорить, что объекты *эквивалентны*, а не идентичны. Использование этих функций STL продемонстрировано в листинге 16.9.

Листинг 16.9. `vect3.cpp`

```
// vect3.cpp -- использование функций STL
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
struct Review {
    std::string title;
    int rating;
};
bool operator<(const Review & r1, const Review & r2);
bool worseThan(const Review & r1, const Review & r2);
bool FillReview(Review & rr);
void ShowReview(const Review & rr);
int main()
{
    using namespace std;
    vector<Review> books;
    Review temp;
    while (FillReview(temp))
    books.push_back(temp);
    if (books.size() > 0)
    {
        cout << "Thank you. You entered the following "
             << books.size() << " ratings:\n"
             << "Rating\tBook\n";
        for_each(books.begin(), books.end(), ShowReview);
        sort(books.begin(), books.end());
        cout << "Sorted by title:\nRating\tBook\n";
              // Список книг, отсортированный по названию
        for_each(books.begin(), books.end(), ShowReview);
        sort(books.begin(), books.end(), worseThan);
        cout << "Sorted by rating:\nRating\tBook\n";
              // Список книг, отсортированный по рейтингу
        for_each(books.begin(), books.end(), ShowReview);
        random_shuffle(books.begin(), books.end());
        cout << "After shuffling:\nRating\tBook\n";
              // Список книг после перемешивания
        for_each(books.begin(), books.end(), ShowReview);
    }
    else
        cout << "No entries. ";
    cout << "Bye.\n";
    return 0;
}
```

```
bool operator<(const Review & r1, const Review & r2)
{
   if (r1.title < r2.title)
      return true;
   else if (r1.title == r2.title && r1.rating < r2.rating)
      return true;
   else
      return false;
}
bool worseThan(const Review & r1, const Review & r2)
{
   if (r1.rating < r2.rating)
      return true;
   else
      return false;
}
bool FillReview(Review & rr)
{
   std::cout << "Enter book title (quit to quit): ";
   std::getline(std::cin,rr.title);
   if (rr.title == "quit")
      return false;
   std::cout << "Enter book rating: ";
   std::cin >> rr.rating;
   if (!std::cin)
      return false;
   // Избавиться от остальной части строки ввода
   while (std::cin.get() != '\n')
      continue;
   return true;
}
void ShowReview(const Review & rr)
{
   std::cout << rr.rating << "\t" << rr.title << std::endl;
}
```

Ниже приведен пример запуска программы из листинга 16.9:

```
Enter book title (quit to quit): The Cat Who Can Teach You Weight Loss
Enter book rating: 8
Enter book title (quit to quit): The Dogs of Dharma
Enter book rating: 6
Enter book title (quit to quit): The Wimps of Wonk
Enter book rating: 3
Enter book title (quit to quit): Farewell and Delete
Enter book rating: 7
Enter book title (quit to quit): quit
Thank you. You entered the following 4 ratings:
Rating Book
8       The Cat Who Can Teach You Weight Loss
6       The Dogs of Dharma
3       The Wimps of Wonk
7       Farewell and Delete
Sorted by title:
Rating Book
7       Farewell and Delete
```

```
8       The Cat Who Can Teach You Weight Loss
6       The Dogs of Dharma
3       The Wimps of Wonk
Sorted by rating:
Rating Book
3       The Wimps of Wonk
6       The Dogs of Dharma
7       Farewell and Delete
8       The Cat Who Can Teach You Weight Loss
After shuffling:
Rating Book
7       Farewell and Delete
3       The Wimps of Wonk
6       The Dogs of Dharma
8       The Cat Who Can Teach You Weight Loss
Bye.
```

Цикл `for`, основанный на диапазоне (C++11)

Цикл `for`, основанный на диапазоне, который был упомянут в главе 5, предназначен для работы с библиотекой STL. В качестве напоминания снова обратимся к первому примеру из главы 5:

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (double x : prices)
   cout << x << std::endl;
```

Аргументы, указанные в круглых скобках оператора `for`, объявляют тип переменной, хранящейся в контейнере, и имя этого контейнера. Затем в теле цикла именованная переменная используется для поочередного обращения к каждому из элементов контейнера. Например, рассмотрим следующий оператор из листинга 16.9:

```
for_each(books.begin(), books.end(), ShowReview);
```

Его можно заменить следующим циклом `for`, основанным на диапазоне:

```
for (auto x : books) ShowReview(x);
```

Компилятор будет использовать тип объекта `books`, которым является `vector<Review>`, для определения того, что типом объекта `x` является `Review`, и цикл будет по очереди передавать каждый объект `Review` контейнера `books` в функцию `ShowReview()`.

В отличие от функции `for_each()`, цикл `for`, основанный на диапазоне, может изменять содержимое контейнера. При этом важно указать параметр ссылки. Например, предположим, что имеется следующее определение функции:

```
void InflateReview(Review &r){r.rating++;}
```

Эту функцию можно было бы применить к каждому элементу в объекте `books` с помощью такого цикла:

```
for (auto & x : books) InflateReview(x);
```

Обобщенное программирование

Теперь, когда вы получили некоторый опыт использования STL, рассмотрим философию, лежащую в основе этой библиотеки. STL — это пример *обобщенного программирования*. При объектно-ориентированном программировании основное внимание

уделяется аспекту данных, а при обобщенном — алгоритмам. Общим аспектом этих двух подходов является абстрагирование и создание повторно используемого кода, но положенная в их основу философия довольно отличается.

Цель обобщенного программирования — создание кода, который не зависит от типов данных. Шаблоны — это средства C++, предназначенные для создания обобщенных программ. Естественно, шаблоны позволяют определять функции или классы в терминах обобщенного типа. Библиотека STL продвигается еще дальше, предоставляя обобщенное представление алгоритмов. Шаблоны делают это возможным, но при тщательном и продуманном проектном решении. Чтобы увидеть, как работает эта смесь шаблонов и проектных решений, давайте посмотрим, для чего нужны итераторы.

Предназначение итераторов

Вероятно, понимание работы итераторов — одно из главных условий понимания STL. Подобно тому, как шаблоны обеспечивают независимость алгоритмов от типа хранимых данных, итераторы обеспечивают независимость от типа используемых контейнеров. Таким образом, они являются существенным компонентом обобщенного подхода STL.

Чтобы увидеть, зачем нужны итераторы, взглянем, как можно реализовать функцию `find` для двух различных представлений данных и как затем обобщить подход. Вначале рассмотрим функцию, которая осуществляет поиск некоторого значения в обычном массиве элементов типа `double`. Эту функцию можно определить следующим образом:

```
double * find_ar(double * ar, int n, const double & val)
{
    for (int i = 0; i < n; i++)
        if (ar[i] == val)
            return &ar[i];
    return 0;  // или return nullptr; в C++11
}
```

Если функция находит значение в массиве, она возвращает адрес в массиве, по которому находится указанное значение, или в противном случае нулевой указатель. Для перемещения по массиву она использует нотацию индексов. Чтобы обобщить эту функцию для работы с массивами любого типа, к которым применима операция ==, можно использовать шаблон. Однако этот алгоритм по-прежнему применим только для определенной структуры данных — массива.

Поэтому рассмотрим поиск в другой структуре — связном списке. (В главе 12 связный список используется для реализации класса `Queue`.) Список состоит из связанных между собой структур `Node`:

```
struct Node
{
    double item;
    Node * p_next;
};
```

Предположим, что есть указатель на первый узел списка. Указатель `p_next` в каждом узле списка указывает на следующий узел, а `p_next` последнего элемента установлен в 0. Функцию `find_ll()` можно было бы написать следующим образом:

```
Node* find_ll(Node * head, const double & val)
{
    Node * start;
```

```
    for (start = head; start!= 0; start = start->p_next)
        if (start->item == val)
            return start;
    return 0;
}
```

Как и в предыдущем случае, с помощью шаблона можно обобщить этот алгоритм для списков данных любого типа, поддерживающих операцию ==. Тем не менее, он останется применимым только к одной определенной структуре данных — связному списку.

Если внимательнее рассмотреть подробности реализации, выяснится, что две функции `find` используют различные алгоритмы: одна применяет индексацию массива для перемещения по списку элементов, а другая — сброс значения `start` в `start->p_next`. Но в широком смысле оба алгоритма совпадают: они последовательно сравнивают искомое значение со значением каждого элемента контейнера до тех пор, пока не будет найдено совпадение.

В данном случае целью обобщенного программирования может быть получение единственной функции `find`, которая бы работала с массивами, со связными списками или с любым другим типом контейнера. То есть функция должна быть независимой не только от типа данных, хранящихся в контейнере, но и от структуры данных самого контейнера. Шаблоны обеспечивают обобщенное представление типа данных, хранимых в контейнере. Что нам необходимо — так это обобщенное представление процесса перемещения по элементам контейнера. Итератор и является таким обобщенным представлением.

Какими свойствами должен обладать итератор, чтобы реализовать функцию `find`? Ниже приведен краткий список этих свойств.

- Нужно иметь возможность разыменовывания итератора, чтобы получить доступ к значению, на которое он ссылается. То есть, если `p` — итератор, то должно быть определено выражение `*p`.
- Должна существовать возможность присваивания одного итератора другому. Это значит, что если `p` и `q` — итераторы, то должно быть определено выражение `p = q`.
- Должна существовать возможность сравнения одного итератора с другим. То есть, если `p` и `q` — итераторы, то должны быть определены выражения `p == q` и `p != q`.
- Должна существовать возможность перемещения итератор по элементам контейнера. Это условие можно удовлетворить, определяя для итератора `p` выражения `++p` и `p++`.

Итератор может решать и другие задачи, но ни одна из них не является обязательной — по крайней мере, для реализации функции `find`. В действительности STL определяет несколько уровней итераторов с возрастающими возможностями, и мы вернемся к этой теме позднее. Кстати, следует отметить, что обычный указатель соответствует требованиям, предъявляемым к итераторам. А потому функцию `find_arr()` можно переписать следующим образом:

```
typedef double * iterator;
iterator find_ar(iterator ar, int n, const double & val)
{
    for (int i = 0; i < n; i++, ar++)
        if (*ar == val)
            return ar;
    return 0;
}
```

Затем список параметров функции можно изменить так, чтобы в качестве аргументов, задающих диапазон, она принимала указатель на начало массива и указатель на элемент, следующий за концом массива. (Нечто подобное выполняется в листинге 7.8 из главы 7.) Функция может возвращать конечный указатель в качестве признака того, что значение не найдено. Эти изменения отражены в следующей версии `find_arr()`:

```
typedef double * iterator;
iterator find_ar(iterator begin, iterator end, const double & val)
{
   iterator ar;
   for (ar = begin; ar != end; ar++)
      if (*ar == val)
         return ar;
   return end;  // признак того, что значение не найдено
}
```

Для функции `find_ll()` можно определить класс итератора, в котором определены операции * и ++:

```
struct Node
{
   double item;
   Node * p_next;
};
class iterator
{
   Node * pt;
public:
   iterator() : pt(0) {}
   iterator (Node * pn) : pt(pn) {}
   double operator*() { return pt->item;}
   iterator& operator++()    // для ++it
   {
      pt = pt->p_next;
      return *this;
   }
   iterator operator++(int)  // для it++
   {
      iterator tmp = *this;
      pt = pt->p_next;
      return tmp;
   }
   // ... operator==(), operator!=() и т.д.
};
```

(Для различения префиксной и постфиксной версий операции ++ в C++ принято соглашение о том, что `operator++()` — это префиксная форма, а `operator++(int)` — постфиксная; аргумент никогда не используется, поэтому в именовании не нуждается.)

В данном случае важно не то, как именно реализован класс `iterator`, а то, что при его использовании вторая функция `find` может быть переписана следующим образом:

```
iterator find_ll(iterator head, const double & val)
{
   iterator start;
   for (start = head; start!= 0; ++start)
      if (*start == val)
         return start;
   return 0;
}
```

Эта функция очень похожа на `find_ar()`. Различие между ними только в том, как обе функции определяют достижение конца списка значений при поиске. Функция `find_ar()` использует итератор, указывающий на позицию, которая расположена за последним элементом, в то время как `find_ll()` работает с нулевым значением, сохраненным в последнем узле. Если устранить это различие, упомянутые функции можно сделать идентичными. Например, можно было бы потребовать, чтобы связный список содержал один дополнительный элемент за последним реальным элементом. То есть можно было бы обеспечить, чтобы и массив, и связный список содержали элемент, расположенный "за последним элементом", и поиск можно было бы завершать по достижении итератором этого элемента. Тогда и `find_arr()`, и `find_ll()` одинаково определяли бы конец данных, и их алгоритмы поиска стали бы идентичными. Следует отметить, что требование наличия дополнительного элемента, расположенного за последним элементом, вытекает из требований к итераторам, которые, в свою очередь, предъявляют требования к классу контейнеров.

Библиотека STL придерживается описанного подхода. Первым делом, каждый контейнерный класс (`vector`, `list`, `deque` и т.д.) определяет тип итератора, соответствующий данному классу. Для одного класса итератор может быть указателем, для другого — объектом. Но какой бы ни была реализация, каждый итератор будет предоставлять необходимые операции — вроде * и ++. (Некоторые классы нуждаются в большем количестве операций, чем другие.) Кроме того, каждый контейнерный класс имеет маркер элемента, расположенного за последним, который представляет собой значение, присваиваемое итератору при выходе за последнее значение контейнера. Каждый класс контейнера имеет методы `begin()` и `end()`, которые возвращают итераторы, соответственно указывающие на первый элемент и на элемент, расположенный за последним. И каждый класс контейнера будет иметь операцию ++, перемещающую итератор от первого элемента до элемента, расположенного за последним, по пути посещая каждый из элементов контейнера.

Для использования класса контейнера не требуется знать ни как реализованы итераторы, ни как реализован элемент, расположенный за последним. Достаточно знать, что у него имеются итераторы, и то, что `begin()` возвращает итератор, указывающий на первый элемент, а `end()` — итератор, который указывает на элемент, расположенный за последним. Например, предположим, что требуется напечатать значения из объекта `vector<double>`. В этом случае можно воспользоваться следующим кодом:

```
vector<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
   cout << *pr << endl;
```

Приведенная ниже строка идентифицирует `pr` как тип итератора, определенного для класса `vector<double>`:

```
vector<double>::iterator pr;
```

Если вместо этого для хранения счетов использовать шаблон класса `list<double>`, можно написать такой код:

```
list<double>::iterator pr;
for (pr = scores.begin(); pr != scores.end(); pr++)
   cout << *pr << endl;
```

Единственное изменение заключается в типе, объявленном для `pr`. То есть, определяя для каждого класса соответствующие итераторы и создавая классы в унифицированной манере, STL позволяет писать один и тот же код для контейнеров, которые имеют совершенно разное внутреннее представление.

Автоматическое выведение типа C++11 позволяет еще больше упростить задачу и применять следующий код при работе и с вектором, и со списком:

```
for (auto pr = scores.begin(); pr != scores.end(); pr++)
   cout << *pr << endl;
```

На самом деле по стилевым соображениям лучше избегать непосредственного применения итераторов; вместо этого, если возможно, следует использовать функции STL, такие как `for_each()`, которые самостоятельно позаботятся о нюансах. Можно также воспользоваться циклом `for`, основанным на диапазоне:

```
for (auto x : scores) cout << x << endl;
```

Итак, подведем итоги ознакомления с подходом, принятым в STL. Работа начинается с определения алгоритма обработки контейнера. Его следует выразить максимально обобщенным образом, обеспечивая независимость от типа данных и типа контейнера. Для обеспечения работы обобщенного алгоритма со специфическими случаями определяются итераторы, соответствующие нуждам алгоритма, и закладываются требования в архитектуру контейнеров. Это значит, что базовые свойства итераторов и контейнеров вытекают из требований, заложенных в алгоритм.

Виды итераторов

Разные алгоритмы предъявляют разные требования к итераторам. Например, алгоритм `find` требует определения операции `++`, чтобы итератор мог пошагово проходить весь контейнер. Ему нужен доступ к данным для чтения, но не для записи. (Он просто просматривает данные, не изменяя их.) С другой стороны, обычный алгоритм сортировки требует произвольного доступа, чтобы иметь возможность обмена значениями для двух не соседствующих элементов. Если `iter` — итератор, произвольный доступ можно получить, определив операцию `+`, чтобы можно было написать выражение вроде `iter + 10`. Кроме того, алгоритм сортировки должен иметь возможность как читать, так и записывать данные. В библиотеке STL определены пять видов итераторов и описаны их алгоритмы в терминах необходимых им разновидностей итераторов. Эти пять видов итераторов следующие: входной итератор, выходной итератор, однонаправленный итератор, двунаправленный итератор и итератор произвольного доступа. Например, прототип `find()` выглядит так:

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Это говорит о том, что алгоритм требует входного итератора. Аналогичным образом следующий прототип указывает, что алгоритм сортировки `sort` нуждается в итераторе произвольного доступа:

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
```

Все пять видов итераторов можно разыменовывать (т.е. для них определена операция `*`) и сравнивать на предмет эквивалентности (с помощью операции `==`, возможно, перегруженной) и неэквивалентности (с помощью операции `!=`, возможно, перегруженной). Если два сравниваемых итератора эквивалентны, то разыменование одного должно порождать то же значение, что и разыменование другого. То есть, если выражение

```
iter1 == iter2
```

истинно, то и следующее выражение также истинно:

```
*iter1 == *iter2
```

Конечно, эти свойства остаются истинными и для встроенных операций и указателей, поэтому данные требования могут служить руководством к действию в плане того, что нужно выполнить при перегрузке этих операций для класса итератора. Теперь рассмотрим другие свойства итераторов.

Входные итераторы

Термин *входной* используется с точки зрения программы. То есть информация, поступающая из контейнера в программу, считается входной, подобно поступающей в программу с клавиатуры. Таким образом, *входной итератор* — это тот, который программа может применять для считывания значений из контейнера. В частности, разыменование входного итератора должно позволить программе прочитать значение из контейнера, но это не обязательно означает возможность изменения этого значения. Поэтому алгоритмы, которые требуют входных итераторов — это алгоритмы, которые не изменяют значения, хранящиеся в контейнере.

Входной итератор должен обеспечивать доступ ко всем значениям в контейнере. Он решает эту задачу, поддерживая операцию ++ в префиксной и в постфиксной форме. Если установить входной итератор на первый элемент в контейнере и инкрементировать его вплоть до достижения элемента, расположенного за последним, он будет указывать по одному разу на каждый элемент контейнера. Кстати, нет никакой гарантии, что второй проход по элементам контейнера с помощью входного итератора будет выполнен в той же последовательности. Также не существует гарантий, что после увеличения значения входного итератора на единицу его предыдущее значение останется доступным для разыменования. Поэтому любой алгоритм, построенный на основе входного итератора, должен быть однопроходным и не зависеть от значений итератора из предшествующего прохода или предшествующих значений итератора из того же самого прохода.

Обратите внимание, что входной итератор является однонаправленным; его можно инкрементировать, но нельзя возвращать в предшествующие состояния.

Выходные итераторы

В контексте STL термин *выходной* означает, что итератор используется для передачи информации из программы в контейнер. (Таким образом, вывод программы является вводом для контейнера.) Выходной итератор аналогичен входному за исключением того, что его разыменование гарантированно предоставляет программе возможность изменять значение контейнера, но не читать его. Если возможность записи без возможности чтения кажется вам странной, вспомните, что то же самое касается вывода на дисплей: `cout` может модифицировать поток символов, отправленный на дисплей, но не может читать с дисплея. Библиотека STL обеспечивает достаточно общий инструментарий, чтобы ее контейнеры могли представлять выходные устройства, поэтому такая же ситуация может возникнуть и при работе с обычными контейнерами. К тому же, если алгоритм модифицирует содержимое контейнера (например, генерируя новые значения, которые должны сохраняться) без чтения этого содержимого, нет причин требовать, чтобы он использовал итератор, способный считывать это содержимое.

Короче говоря, входной итератор можно применять для алгоритмов однократного прохода с доступом только для чтения, а выходной итератор — для алгоритмов однократного прохода с доступом только для записи.

Однонаправленные итераторы

Как входные и выходные итераторы, *однонаправленные* итераторы используют только операцию ++ для навигации по контейнеру. Таким образом, однонаправленный ите-

ратор может осуществлять перемещение по контейнеру только вперед, поэлементно. Однако, в отличие от входных и выходных итераторов, при каждом использовании он обязательно выполняет проход по последовательности значений в одном и том же порядке. Кроме того, после инкрементирования однонаправленного итератора предыдущее значение по-прежнему можно разыменовать, если оно было сохранено, и получить при этом то же самое значение. Эти свойства позволяют реализовать многопроходные алгоритмы. Однонаправленный итератор может позволить как чтение, так и модификацию данных, либо только чтение:

```
int * pirw;              // итератор чтения и записи
const int * pir;         // итератор только для чтения
```

Двунаправленные итераторы

Предположим, что имеется алгоритм, которому нужна возможность прохода контейнера в обоих направлениях. Например, функция обратного прохода может обменивать значения первого и последнего элемента, инкрементировать указатель на первый элемент, декрементировать указатель на второй элемент, а затем повторять этот процесс. *Двунаправленный* итератор обладает всеми свойствами однонаправленного итератора и добавляет к ним поддержку двух операций декремента (префиксной и постфиксной).

Итераторы произвольного доступа

Некоторые алгоритмы, такие как стандартная сортировка и бинарный поиск, требуют возможности перехода непосредственно на произвольный элемент контейнера. Этот способ доступа называется *произвольным доступом* и требует итератора произвольного доступа. Итератор такого типа обладает всеми свойствами двунаправленного итератора, а также операциями (наподобие операции сложения с указателем), которые поддерживают произвольный доступ, и операциями отношения для упорядочения элементов. В табл. 16.3 перечислены операции итераторов произвольного доступа, которые добавлены к нему в дополнение к присущим двунаправленным итераторам. В этой таблице `X` представляет тип итератора произвольного доступа, `T` — тип, на который он указывает, `a` и `b` — значения итератора, а `r` — переменная или ссылка итератора произвольного доступа.

Таблица 16.3. Операции итераторов произвольного доступа

Выражение	Описание
`a + n`	Указывает на `n`-й элемент после того, на который указывает `a`
`n + a`	То же самое, что `a + n`
`a - n`	Указывает на `n`-й элемент перед тем, на который указывает `a`
`r += n`	Эквивалентно `r = r + n`
`r -= n`	Эквивалентно `r = r - n`
`a[n]`	Эквивалентно `*(a + n)`
`b - a`	Такое значение `n`, при котором `b = a + n`
`a < b`	Истинно, если `b - a > 0`
`a > b`	Истинно, если `b < a`
`a >= b`	Истинно, если `!(a < b)`
`a <= b`	Истинно, если `!(a > b)`

Выражения вроде a + n корректны, только если a и a + n лежат в диапазоне контейнера (включая элемент, расположенный за последним).

Иерархия итераторов

Возможно, вы обратили внимание, что виды итераторов образуют иерархию. Однонаправленный итератор обладает всеми свойствами входного и выходного итераторов, а также собственными возможностями. Двунаправленный итератор имеет все свойства однонаправленного итератора плюс собственные. А итератор произвольного доступа имеет все свойства однонаправленного итератора, к которым добавлены его собственные возможности. Основные свойства итераторов обобщены в табл. 16.4. В этой таблице i — итератор, а n — целое значение.

Таблица 16.4. Возможности итераторов

Возможность итератора	Входной	Выходной	Однонаправленный	Двунаправленный	Произвольного доступа
Разыменующее чтение	Да	Нет	Да	Да	Да
Разыменующая запись	Нет	Да	Да	Да	Да
Фиксированный и повторяющийся порядок	Нет	Нет	Да	Да	Да
++i i++	Да	Да	Да	Да	Да
--i i--	Нет	Нет	Нет	Да	Да
i[n]	Нет	Нет	Нет	Нет	Да
i + n	Нет	Нет	Нет	Нет	Да
i – n	Нет	Нет	Нет	Нет	Да
i += n	Нет	Нет	Нет	Нет	Да
i -= n	Нет	Нет	Нет	Нет	Да

Алгоритмы, написанные в терминах определенного вида итераторов, могут использовать этот итератор или любой другой, обладающий нужными возможностями. Поэтому, например, контейнер с итератором произвольного доступа может пользоваться алгоритмом, написанным для входного итератора.

Зачем нужны все эти разные виды итераторов? Идея состоит в том, чтобы написать алгоритм, использующий итератор с минимально необходимыми требованиями, что позволит его применять с максимальным множеством различных контейнеров. Так, функция find(), за счет использования входного итератора начального уровня, может применяться с любым контейнером, который содержит читаемые значения. Однако функция sort(), которая требует итераторов произвольного доступа, может использоваться только с контейнерами, поддерживающими этот вид итераторов.

Обратите внимание, что различные виды итераторов не являются определенными типами. Скорее, они представляют собой концептуальные характеристики. Как упоминалось ранее, каждый класс контейнера определяет typedef с областью видимости класса по имени. Поэтому класс vector<int> имеет итератор типа vector<int>::iterator. Но в документации по этому классу утверждается, что итераторы вектора — это итераторы произвольного доступа. В свою очередь, данный факт позволяет применять алгоритмы, базирующиеся на итераторах любого типа, потому что итератор произвольного доступа обладает возможностями всех итера-

торов. Аналогично класс list<int> имеет итераторы типа list<int>::iterator. Библиотека STL реализует двунаправленные связные списки, которые используют двунаправленный итератор. Поэтому такой список не может применять алгоритмы на основе итераторов произвольного доступа, но может использовать алгоритмы на базе менее требовательных итераторов.

Концепции, уточнения и модели

Библиотека STL имеет некоторые средства, такие как виды итераторов, которые нельзя выразить на языке C++. То есть, хотя можно, например, спроектировать класс, обладающий свойствами однонаправленного итератора, компилятор нельзя ограничить алгоритмом, использующим только этот класс. Причина в том, что однонаправленный итератор — это набор требований, а не тип. Требования могут быть удовлетворены специально спроектированным классом итератора, но они могут также быть удовлетворены и обычным указателем. Алгоритм STL работает с любой реализацией итератора, которая соответствует его требованиям. В литературе по STL для описания набора требований применяется слово *концепция*. Таким образом, существует концепция входного итератора, концепция однонаправленного итератора и т.д. Кстати говоря, если, например, нужны итераторы для конкретного проектируемого класса контейнера, можно обратиться к библиотеке STL, которая включает шаблоны итераторов всех стандартных разновидностей.

Концепции могут быть связаны между собой отношением, подобным наследованию. Например, двунаправленный итератор наследует возможности однонаправленного итератора. Однако механизм наследования C++ нельзя применить к итераторам. Например, однонаправленный итератор можно реализовать как класс, а двунаправленный — как обычный указатель. Таким образом, с точки зрения C++ данный конкретный двунаправленный итератор, будучи встроенным типом, не может быть производным от класса. Однако концептуально он его наследует. В некоторых публикациях по STL для указания этого концептуального наследования применяется термин *уточнение* (refinement). Таким образом, двунаправленный итератор — это уточнение концепции однонаправленного итератора.

Конкретная реализация концепции называется *моделью*. Так, обычный указатель на целое — это модель концепции итератора произвольного доступа. Он является также моделью однонаправленного итератора, поскольку удовлетворяет всем требованиям этой концепции.

Указатель как итератор

Итераторы — это обобщения указателей, и указатели отвечают всем требованиям, предъявляемым к итераторам. Итераторы образуют интерфейс для алгоритмов STL, а указатели являются итераторами, поэтому алгоритмы STL могут использовать указатели для работы с не относящимися к STL контейнерами, которые построены на основе указателей. Например, алгоритмы STL можно применять к массивам. Предположим, что Receipts — массив значений типа double, который нужно отсортировать в порядке возрастания:

```
const int SIZE = 100;
double Receipts[SIZE];
```

Вспомним, что функция sort() из STL принимает в качестве аргументов итераторы, указывающие на первый элемент контейнера, и итератор, указывающий на элемент, следующий за последним. Итак, &Receipts[0] (или просто Receipts) — это адрес первого элемента, а &Receipts[SIZE] (или просто Receipts + SIZE) — это ад-

рес элемента, следующего за последним элементом массива. Тогда следующий вызов функции выполняет сортировку массива:

```
sort(Receipts, Receipts + SIZE);
```

C++ гарантирует, что выражение `Receipts + n` определено до тех пор, пока результат лежит в пределах массива или располагается на один элемент дальше его конца. Таким образом, C++ поддерживает концепцию элемента, расположенного за последним, для указателей внутри массива, и это позволяет применять алгоритмы STL к обычным массивам. То есть тот факт, что указатели являются итераторами, а алгоритмы построены на основе итераторов, делает возможным применение алгоритмов STL к обычным массивам. Аналогично, алгоритмы STL можно применять к формам данных собственной разработки при условии обеспечения соответствующих итераторов (которые могут быть указателями или объектами), а также индикаторов элементов, расположенных за последним.

`copy()`, `ostream_iterator` и `istream_iterator`

Библиотека STL предоставляет ряд заранее определенных итераторов. Чтобы понять, с чем это связано, ознакомимся с рядом основополагающих положений. Существует алгоритм `copy()`, предназначенный для копирования данных из одного контейнера в другой. Этот алгоритм выражен в терминах итераторов, поэтому он может копировать из одного вида контейнеров в другой, и даже из массива или в него, поскольку в качестве итераторов можно применять указатели массива. Например, следующий код копирует массив в вектор:

```
int casts[10] = {6, 7, 2, 9 ,4 , 11, 8, 7, 10, 5};
vector<int> dice[10];
copy(casts, casts + 10, dice.begin());   // копирование массива в вектор
```

Первые два аргумента-итератора функции `copy()` представляют диапазон, который следует скопировать, а последний аргумент-итератор — местоположение, куда копируется первый элемент. Первые два аргумента должны быть входными (или более совершенными) итераторами, а заключительный аргумент — выходным (или более совершенными) итератором. Функция `copy()` переписывает существующие данные в контейнере назначения, и этот контейнер должен быть достаточно велик, чтобы вместить копируемые элементы. Поэтому функцию `copy()` нельзя использовать для помещения данных в пустой вектор — по крайней мере, не прибегнув к ухищрению, описанному далее в этой главе.

Теперь предположим, что требуется копировать информацию на дисплей. Для этого можно было бы воспользоваться функцией `copy()`, если бы существовал итератор, представляющий выходной поток. Шаблон `ostream_iterator` из STL предоставляет такой итератор. Согласно терминологии STL, этот шаблон представляет собой *модель* концепции выходного итератора. Одновременно он является примером *адаптера* — класса или функции, которая преобразует какой-то другой интерфейс в интерфейс, используемый STL. Итератор этого вида можно создать, включив заголовочный файл `iterator` (ранее — `iterator.h`) и сделав следующее объявление:

```
#include <iterator>
ostream_iterator<int, char> out_iter(cout, " ");
```

Теперь итератор `out_iter` становится интерфейсом, который позволяет применять `cout` для отображения. Первый аргумент шаблона (в данном случае `int`) указывает тип данных, отправляемых в выходной поток. Второй аргумент шаблона (в этом случае `char`) задает символьный тип, используемый выходным потоком (другим до-

пустимым значением могло бы быть wchar_t). Первый аргумент конструктора (в рассматриваемой ситуации это cout) идентифицирует используемый выходной поток. Им мог бы быть также поток, используемый для вывода файла. Последний строковый аргумент — это разделитель, который должен отображаться после каждого элемента, отправленного в выходной поток. Итератор можно было бы применять следующим образом:

```
*out_iter++ = 15;  // работает подобно cout << 15 << " ";
```

Для обычного указателя это означало бы присваивание значения 15 переменной, находящейся по адресу out_iter, с последующим инкрементированием этого указателя. Однако для данного итератора ostream_iterator приведенный оператор означает отправку значения 15 и строки, состоящей из пробела, в выходной поток, управляемый cout. После этого выходной поток должен быть готов к следующей операции вывода. С функцией copy() итератор можно было бы использовать следующим образом:

```
copy(dice.begin(), dice.end(), out_iter); //копирование вектора в выходной поток
```

Это означало бы копирование всего содержимого контейнера dice в выходной поток, т.е. отображение содержимого контейнера.

Или же можно пропустить создание именованного итератора и вместо него сконструировать неименованный итератор. Это значит, что можно воспользоваться примерно таким адаптером:

```
copy(dice.begin(), dice.end(), ostream_iterator<int, char>(cout, " "));
```

Аналогично, заголовочный файл iterator определяет шаблон istream_iterator для адаптации ввода istream к интерфейсу входного итератора. Это — модель концепции входного итератора. Для определения входного диапазона функции copy() можно применять два объекта istream_iterator:

```
copy(istream_iterator<int, char>(cin),
    istream_iterator<int, char>(), dice.begin());
```

Подобно ostream_iterator, объект istream_iterator использует два аргумента шаблона. Первый указывает тип данных, который будет прочитан, а второй — символьный тип, используемый входным потоком. Применение аргумента конструктора cin означает использование входного потока, управляемого cin. Опускание аргумента конструктора означает ошибку ввода, поэтому приведенный выше код означает чтение из входного потока вплоть до возникновения условия конца файла, несоответствия типа или другой ошибки ввода.

Другие полезные итераторы

В дополнение к ostream_iterator и istream_iterator, заголовочный файл iterator предоставляет и другие заранее определенные итераторы специального назначения, среди которых reverse_iterator, back_insert_iterator, front_insert_iterator и insert_iterator.

Начнем с рассмотрения того, что делает обратный итератор (reverse_iterator). В сущности, инкрементирование этого итератора вызывает его декремент. Почему бы просто не применить декремент к обычному итератору? Главная причина — упрощение использования существующих функций. Предположим, что требуется отобразить содержимое контейнера dice. Как вы только что видели, можно воспользоваться copy() и ostream_iterator для копирования содержимого в выходной поток:

```
ostream_iterator<int, char> out_iter(cout, " ");
copy(dice.begin(), dice.end(), out_iter);  // отображение в прямом порядке
```

Теперь предположим, что требуется вывести элементы в обратном порядке. (Возможно, для проведения ретроспективного исследования.) Существует несколько подходов, которые не работают, но вместо того, чтобы возиться с ними, обратимся к тому, который работает. Класс `vector` имеет функцию-член `rbegin()`, возвращающую обратный итератор, который указывает на элемент, находящийся за последним, и функцию-член `rend()`, возвращающую обратный итератор, который указывает на первый элемент. Поскольку инкрементирование обратного итератора приводит к его декременту, для отображения содержимого в обратном порядке можно применить следующий оператор:

```
copy(dice.rbegin(), dice.rend(), out_iter); // отображение в обратном порядке
```

Обратный итератор даже не нужно объявлять.

На заметку!

И `rbegin()`, и `end()` возвращают одно и то же значение (находящееся за последним элементом), но разного типа (`reverse_iterator` и `iterator`, соответственно). Аналогично, `rend()` и `begin()` возвращают одно и то же значение (итератор, указывающий на первый элемент), но разного типа.

Обратные указатели должны выполнять специальную компенсацию. Предположим, что `rp` — обратный указатель, инициализированный в `dice.rbegin()`. Каким должно быть значение `*rp`? Поскольку `rbegin()` возвращает элемент, находящийся за последним в контейнере, не нужно пытаться разыменовывать этот адрес. Аналогично, если `rend()` — действительное местоположение первого элемента, `copy()` останавливается за один элемент до первого элемента контейнера, поскольку диапазон не включает в себя последний элемент. Обратные указатели решают обе проблемы, вначале выполняя декремент, а затем разыменование. То есть `*rp` разыменовывает значение итератора, непосредственно предшествующее текущему значению `*rp`. Если `rp` указывает на шестую позицию в контейнере, то `*rp` — значение из пятой позиции, и т.д. Применение функции `copy()`, итератора `ostream` и обратного итератора иллюстрируется в листинге 16.10.

Листинг 16.10. `copyit.cpp`

```
// copyit.cpp -- copy() и итераторы
#include <iostream>
#include <iterator>
#include <vector>

int main()
{
    using namespace std;

    int casts[10] = {6, 7, 2, 9 ,4 , 11, 8, 7, 10, 5};
    vector<int> dice(10);

    // Копирование из массива в вектор
    copy(casts, casts + 10, dice.begin());
    cout << "Let the dice be cast!\n";

    // Создание итератора ostream
    ostream_iterator<int, char> out_iter(cout, " ");

    // Копирование из вектора в выходной поток
    copy(dice.begin(), dice.end(), out_iter);
    cout << endl;
```

```
    cout <<"Implicit use of reverse iterator.\n";
         // неявное использование обратного итератора
    copy(dice.rbegin(), dice.rend(), out_iter);
    cout << endl;
    cout <<"Explicit use of reverse iterator.\n";
         // явное использование обратного итератора
    vector<int>::reverse_iterator ri;
    for (ri = dice.rbegin(); ri != dice.rend(); ++ri)
       cout << *ri << ' ';
    cout << endl;
    return 0;
}
```

Вывод программы из листинга 16.10 выглядит следующим образом:

```
Let the dice be cast!
6 7 2 9 4 11 8 7 10 5
Implicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
Explicit use of reverse iterator.
5 10 7 8 11 4 9 2 7 6
```

Когда есть возможность выбора между явным объявлением итераторов и использованием функций STL для выполнения всей работы внутренним образом, например, посредством передачи возвращаемого значения `rbegin()` функции, следует отдавать предпочтение второму подходу. В этом случае придется делать меньше работы и будет меньше шансов допустить ошибку.

Другие три итератора (`back_insert_iterator`, `front_insert_iterator` и `insert_iterator`) также повышают степень обобщенности алгоритмов STL. Многие функции STL подобны `copy()` в том, что передают свои результаты по адресу, указанному выходным итератором. Вспомните, что следующий оператор копирует значения в позицию, начинающуюся с `dice.begin()`:

```
copy(casts, casts + 10, dice.begin());
```

Эти значения замещают предыдущее содержимое `dice`, и функция предполагает, что `dice` имеет достаточно места, чтобы уместить все значения. Это значит, что `copy()` не предпринимает автоматическое изменение размеров контейнера назначения, чтобы вместить переданную в него информацию. Программа в листинге 16.10 заботится об этом, объявляя `dice` с 10 элементами, но предположим, что требуемый размер `dice` заранее не известен. Или представим, что требуется добавлять элементы в `dice`, а не замещать существующие.

Эти проблемы решают три итератора вставки, преобразуя процесс копирования в процесс вставки. Вставка добавляет новые элементы, не перезаписывая существующие данные, и при этом использует автоматическое выделение памяти для обеспечения того, что новая информация поместится в контейнере.

Итератор `back_insert_iterator` вставляет элементы в конец контейнера, а `front_insert_iterator` — в его начало. И, наконец, `insert_iterator` вставляет элементы перед позицией, указанной в аргументе конструктора `insert_iterator`. Все эти три итератора являются моделями концепции выходного контейнера.

Однако существуют некоторые ограничения. Так, например, `back_insert_iterator` может применяться только с контейнерными типами, которые допускают быструю вставку в конец. (Термин *быстрая* относится к алгоритму с постоянным временем; понятие постоянного времени рассматривается в разделе "Концепции контейнеров" далее в главе.) Класс `vector` позволяет это делать.

Итератор front_insert_iterator может использоваться только с контейнерными типами, допускающими вставку в начало за постоянное время. Класс vector не позволяет это делать, а класс queue — позволяет. Итератор insert_iterator не обладает этими ограничениями, т.е. его можно применять для вставки данных в начало вектора. Однако front_insert_iterator выполняет это быстрес с теми контейнерными типами, которые поддерживают его применение.

Совет

Итератор insert_iterator можно использовать для преобразования алгоритма, который копирует данные, в алгоритм, вставляющий их.

Эти итераторы принимают тип контейнера в качестве аргумента шаблона и фактический идентификатор контейнера — как аргумент конструктора. Таким образом, чтобы создать итератор back_insert_iterator для контейнера vector<int> по имени dice, нужно написать следующий оператор:

```
back_insert_iterator<vector<int> > back_iter(dice);
```

Необходимость объявления типа контейнера обусловлена тем, что итератор должен применять соответствующий метод контейнера.

Код конструктора back_insert_iterator будет исходить из предположения о существовании метода push_back() для переданного ему типа. Функция copy(), будучи автономной функцией, не обладает правами доступа для изменения размера контейнера. Но приведенное объявление позволяет back_iter пользоваться методом vector<int>::push_back(), который обладает нужными правами доступа.

Объявление front_insert_iterator имеет ту же форму. Объявление insert_iterator содержит дополнительный аргумент конструктора для указания позиции вставки:

```
insert_iterator<vector<int> > insert_iter(dice, dice.begin() );
```

Код в листинге 16.11 иллюстрирует применение этих двух итераторов. Кроме того, для вывода вместо итератора ostream он использует функцию for_each().

Листинг 16.11. inserts.cpp

```
// inserts.cpp -- copy() и итераторы вставки
#include <iostream>
#include <string>
#include <iterator>
#include <vector>
#include <algorithm>

void output(const std::string & s) {std::cout << s << " ";}

int main()
{
    using namespace std;
    string s1[4] = {"fine", "fish", "fashion", "fate"};
    string s2[2] = {"busy", "bats"};
    string s3[2] = {"silly", "singers"};
    vector<string> words(4);
    copy(s1, s1 + 4, words.begin());
    for_each(words.begin(), words.end(), output);
    cout << endl;

    // Конструирование анонимного объекта типа back_insert_iterator
    copy(s2, s2 + 2, back_insert_iterator<vector<string> >(words));
```

```
    for_each(words.begin(), words.end(), output);
    cout << endl;

    // Конструирование анонимного объекта типа insert_iterator
    copy(s3, s3 + 2, insert_iterator<vector<string> >(words, words.begin()));
    for_each(words.begin(), words.end(), output);
    cout << endl;
    return 0;
}
```

Вывод программы из листинга 16.11 имеет следующий вид:

```
fine fish fashion fate
fine fish fashion fate busy bats
silly singers fine fish fashion fate busy bats
```

Первый вызов `copy()` копирует четыре строки из `s1` в `words`. Этот вызов работает, в частности, и потому, что массив `words` объявлен как содержащий четыре строки, что равно количеству копируемых строк. Затем `back_insert_iterator` вставляет строки из `s2` в место, находящееся перед концом массива `words`, увеличивая размер `words` до шести элементов. И, наконец, `insert_iterator` вставляет две строки из `s3` перед первым элементом `words`, увеличивая его размер до восьми элементов. Если бы программа попыталась копировать `s2` и `s3` в массив `words`, используя функции `words.end()` и `words.begin()` в качестве итераторов, скорее всего, в массиве `words` не хватило бы места для новых данных и программа, вероятно, была бы прервана вследствие нарушений, связанных с памятью.

Несмотря на обилие вариаций итераторов, имейте в виду, что их практическое применение поможет быстро освоиться с ними. Также помните, что эти заранее определенные итераторы повышают степень обобщения алгоритмов STL. Так, `copy()` позволяет копировать информацию не только из одного контейнера в другой, но и из контейнера в выходной поток и из входного потока в контейнер. `copy()` можно применять также для вставки данных в другой контейнер. Таким образом, единственная функция может выполнять работу многих других. И поскольку `copy()` — лишь одна из нескольких функций STL, которые используют выходной итератор, эти заранее определенные итераторы увеличивают возможности и этих функций.

Виды контейнеров

Библиотека STL включает в себя как концепции контейнеров, так и типы контейнеров. Концепции — это общие категории с названиями наподобие контейнер, последовательный контейнер и ассоциативный контейнер. Типы контейнеров — это шаблоны, которые можно применять для создания специфических объектов-контейнеров. Изначально были определены 11 типов контейнеров: `deque`, `list`, `queue`, `priority_queue`, `stack`, `vector`, `map`, `multimap`, `set`, `multiset` и `bitset`. (В настоящей главе не рассматривается `bitset` — контейнер для работы с данными на уровне битов.) В C++11 были добавлены типы `forward_list`, `unordered_map`, `unordered_multimap`, `unordered_set` и `unordered_multiset`, а `bitset` перемещен из категории контейнеров в отдельную категорию. Поскольку концепции разделяют типы на категории, начнем с них.

Концепции контейнеров

Концепции элементарного контейнера не соответствует ни один тип, однако эта концепция описывает элементы, общие для всех контейнерных классов. Она представляет собой разновидность концептуального абстрактного класса — концептуально-

го потому, что классы контейнеров на самом деле не используют механизм наследования. Или, если посмотреть на это иначе, концепция контейнера устанавливает набор требований, которым должны удовлетворять все классы контейнеров STL.

Контейнер — это объект, который хранит в себе другие объекты одного типа. Хранимые объекты могут быть объектами в смысле объектно-ориентированного программирования либо значениями встроенных типов. Данные, сохраненные в контейнере, *принадлежат* ему. Это означает, что когда время существования контейнера истекает, это же происходит с сохраненными в нем данными. (Однако если данные являются указателями, истечение срока существования указываемых ими данных не обязательно.)

В контейнере нельзя сохранять объекты какого угодно вида. В частности, тип хранимых объектов должен допускать *присваивание* и *конструирование копированием*. Базовые типы удовлетворяют этим требованиям, как и типы классов — если только определение класса не объявляет конструктор копирования и/или операцию присваивания закрытыми или защищенными. (В C++11 эти концепции уточняются за счет добавления таких терминов, как *допускающий вставку копированием* (CopyInsertable) и *допускающий вставку переносом* (MoveInsertable), но мы ограничимся несколько упрощенным, хотя и менее точным описанием.)

Базовый контейнер не гарантирует, что его элементы будут сохранены в каком-то определенном порядке или же что этот порядок не изменится, но подобные гарантии могут быть получены за счет уточнений концепции. Все контейнеры предоставляют определенные функциональные возможности и операции. Некоторые из этих общих функциональных возможностей кратко описаны в табл. 16.5. В таблице посредством `X` представляется тип контейнера (такой как `vector`), `T` — тип объекта, хранящегося в контейнере, `a` и `b` — значения типа `X`, `r` — значение типа `X&`, а `u` — идентификатор типа `X` (т.е. если `X` представляет `vector<int>`, то `u` — это объект `vector<int>`).

Таблица 16.5. Некоторые основные свойства контейнеров

Выражение	Возвращаемый тип	Описание	Сложность
`X::iterator`	Тип итератора, указывающего на `T`	Итератор любой категории, удовлетворяющий требованиям к однонаправленному итератору	Время компиляции
`X::value_type`	`T`	Тип для `T`	Время компиляции
`X u;`		Создает пустой контейнер с именем `u`	Постоянная
`X();`		Создает пустой анонимный (без имени) контейнер	Постоянная
`X u(a);`		Пост-условие конструктора копирования: `u == a`	Линейная
`X u = a;`		Оказывает тот же эффект, что и `X u(a);`	Линейная
`r = a;`	`X&`	Постусловие присваивания копированием: `r == a`	Линейная
`(&a)->~X();`	`void`	Применяет деструктор к каждому элементу контейнера	Линейная

Окончание табл. 16.5

Выражение	Возвращаемый тип	Описание	Сложность
`a.begin()`	`iterator`	Возвращает итератор, указывающий на первый элемент контейнера	Постоянная
`a.end()`	`iterator`	Возвращает итератор, указывающий на значение, расположенное за последним элементом контейнера	Постоянная
`a.size()`	Беззнаковый целочисленный тип	Возвращает число элементов, равное `a.end() - a.begin()`	Постоянная
`a.swap(b)`	`void`	Обменивает значения `a` и `b`	Постоянная
`a == b`	Преобразуемый в `bool`	Возвращает `true`, если `a` и `b` имеют одинаковый размер и каждый элемент `a` эквивалентен соответствующему элементу `b` (т.е. операция `==` возвращает значение `true`)	Линейная
`a != b`	Преобразуемый в `bool`	Возвращает `!(a == b)`	Линейная

Столбец "Сложность" в табл. 16.5 описывает время, необходимое для выполнения операции. В таблице встречаются три возможных обозначения — от самой быстрой операции до самой медленной:

- время компиляции;
- постоянная;
- линейная.

Если в столбце "Сложность" указано "Время компиляции", это означает, что действие выполняется во время компиляции и не требует времени при выполнении. Постоянная сложность означает, что операция происходит во время выполнения, но не зависит от количества элементов в объекте. Линейная сложность значит, что время операции пропорционально количеству элементов. То есть, если `a` и `b` являются контейнерами, то сравнение `a == b` имеет линейную сложность, потому что операция `==` должна быть применена к каждому элементу контейнера. На самом деле — это худший сценарий. Если два контейнера имеют разный размер, то никаких индивидуальных сравнений элементов выполнять не требуется.

Сложность, описываемая постоянным и линейным временем

Вообразите длинный и узкий ящик, наполненный большими пакетами, которые уложены в линейку, причем этот ящик открыт только с одной стороны. Предположим, что ваша задача — выгрузить пакет из открытой стороны. Это задача, выполнение которой потребует постоянного времени, которое не зависит от того, сколько пакетов находится за последним — 10 или 1000.

Теперь представьте, что вашей задачей является извлечение пакета, расположенного с закрытой стороны ящика. Это задача, время выполнения которой изменяется линейно. Когда в ящике всего 10 пакетов, чтобы добраться до последнего, придется выгрузить 10 пакетов. Если их 100, то придется выгрузить все 100. Если считать, что работу выполняет неутомимый работник, который может перемещать только по одному пакету за раз, выполнение этой задачи займет в 10 раз больше времени, чем первой.

Теперь предположим, что требуется извлечь произвольный пакет. Может оказаться, что он будет первым. Однако в среднем количество пакетов, которые придется переместить, по-прежнему пропорционально общему числу пакетов в контейнере, поэтому сложность этой задачи также линейна с точки зрения времени выполнения.

Замена длинного и узкого ящика подобным, но имеющим открывающиеся боковые стенки, делает время, необходимое для выполнения этого задания, постоянным, поскольку, открыв боковую стенку ящика, можно обратиться непосредственно к нужному пакету и вытащить его, не трогая остальных.

Идея сложности с точки зрения времени выполнения описывает влияние размера контейнера на время выполнения операций, но игнорирует другие факторы. Если некий супергерой может выгружать ящики из открытой стороны в 1000 раз быстрее, нежели вы, задача по-прежнему будет иметь линейную сложность, но в этом случае линейное время доступа супергероя к пакетам закрытого ящика (с одной открытой стороной) может оказаться меньше, чем ваше постоянное время доступа к пакетам открытого ящика, если только пакетов в ящиках не слишком много.

Требования к сложности — это характеристика STL. Хотя нюансы реализации могут быть скрыты, характеристики производительности должны быть открытыми, чтобы можно было оценить вычислительные затраты на выполнение конкретной операции.

Дополнения C++11 к требованиям, связанным с контейнерами

В табл. 16.6 приведены некоторые дополнения C++11, добавленные к общим требованиям к контейнерам. В этой таблице `rv` используется для обозначения не являющегося константой значения rvalue типа `X` (например, возвращаемого значения функции). Кроме того, согласно табл. 16.6, требование соответствия `X::iterator` однонаправленному итератору отличается от первоначального только тем, что данный итератор не является выходным итератором.

Таблица 16.6. Некоторые дополнения к основным требованиям к контейнерам (C++11)

Выражение	Возвращаемый тип	Описание	Сложность
`X u(rv);`		Постусловие конструктора переноса: `u` содержит значение, которое было у `rv` до конструирования	Линейная
`X u = rv;`		Оказывает тот же эффект, что и `X u(rv);`	
`a = rv;`	`X&`	Постусловие присваивания переносом: `a` содержит значение, которое было у `rv` до присваивания	Линейная
`a.cbegin()`	`const_iterator`	Возвращает итератор `const`, ссылающийся на первый элемент контейнера	Постоянная
`a.cend()`	`const_iterator`	Возвращает итератор `const`, представляющий значение, которое расположено за последним элементом контейнера	Постоянная

Различие между конструированием с помощью копирования и присваиванием с помощью копирования с одной стороны, и конструированием посредством переноса и присваиванием посредством переноса с другой состоит в том, что операция копирова-

ния оставляет исходное значение неизменным, а операция переноса может изменять исходное значение, возможно, перенося права владения без выполнения какого-либо копирования. Когда исходный объект является временным, операции переноса могут предоставить более эффективный код, чем обычное копирование. Семантика переноса подробнее рассматривается в главе 18.

Последовательности

Базовую концепцию контейнера можно уточнять, добавляя требования. *Последовательность* — это важное уточнение, поскольку несколько типов контейнеров STL — `deque`, `forward_list` (C++11), `list`, `queue`, `priority_queue`, `stack` и `vector` — являются последовательностями. (Вспомните, что очередь (queue) позволяет добавлять элементы в конце и удалять в начале. Двусторонняя очередь (double-ended queue), представленная контейнером `deque`, допускает добавление и извлечение с обеих сторон.) Уточнение, заключающееся в том, что итератор должен быть, по меньшей мере, однонаправленным, гарантирует размещение элементов в определенном порядке, который не меняется от одного цикла итераций к другому. Класс `array` также считается контейнером очереди, хотя он удовлетворяет не всем требованиям.

Последовательность также требует, чтобы элементы были упорядочены в строго линейном порядке. Другими словами, есть первый элемент, а также последний элемент, и каждый элемент кроме первого и последнего имеет только один элемент непосредственно перед ним и только один — непосредственно за ним. Массив и связный список являются примерами последовательностей, в то время как структуры ветвления (в которых каждый узел указывает на два дочерних) — нет.

Поскольку элементы в последовательностях размещены в определенном порядке, становятся возможными такие операции, как вставка значений в определенную позицию и удаление определенного диапазона элементов. В табл. 16.7 перечислены эти и другие операции, необходимые последовательностям. В таблице применяются те же обозначения, что и в табл. 16.5, с добавлением `t`, представляющего значение типа `T` — т.е. типа значений, хранимых в контейнере, а также `n` — целого и `p`, `q`, `i` и `j`, представляющих итераторы.

Таблица 16.7. Требования к последовательностям

Выражение	Возвращаемый тип	Описание
`X a(n,t);`		Объявляет последовательность `a` из `n` копий значения `t`
`X(n, t)`		Создает анонимную последовательность из `n` копий значения `t`
`X a(i, j)`		Объявляет последовательность `a`, инициализированную содержимым из диапазона [`i`, `j`)
`X(i, j)`		Создает анонимную последовательность, инициализированную содержимым из диапазона [`i`, `j`)
`a.insert(p,t)`	итератор	Вставляет копию `t` перед `p`
`a.insert(p,n,t)`	`void`	Вставляет `n` копий `t` перед `p`
`a.insert(p,i,j)`	`void`	Вставляет копии элементов диапазона [`i`, `j`) перед `p`
`a.erase(p)`	итератор	Удаляет элемент, на который указывает `p`
`a.erase(p,q)`	итератор	Удаляет элементы диапазона [`p`, `q`)
`a.clear()`	`void`	То же самое, что и `erase(begin(), end())`

Поскольку классы шаблонов deque, list, priority_queue, stack и vector являются моделями концепции последовательности, все они поддерживают операции из табл. 16.7. В дополнение к этому существуют операции, которые доступны некоторым из упомянутых шести моделей. Когда они разрешены, то имеют постоянное время выполнения. Эти дополнительные операции перечислены в табл. 16.8.

Таблица 16.8. Необязательные требования к последовательностям

Выражение	Возвращаемый тип	Значение	Контейнер
a.front()	T&	*a.begin()	vector, list, deque
a.back()	T&	*--a.end()	vector, list, deque
a.push_front(t)	void	a.insert(a.begin(), t)	list, deque
a.push_back(t)	void	a.insert(a.end(), t)	vector, list, deque
a.pop_front(t)	void	a.erase(a.begin())	list, deque
a.pop_back(t)	void	a.erase(--a.end())	vector, list, deque
a[n]	T&	*(a.begin() + n)	vector, deque
a.at(n)	T&	*(a.begin() + n)	vector, deque

Содержимое табл. 16.8 требует нескольких комментариев. Первым делом, обратите внимание, что и a[n], и a.at(n) возвращают ссылку на n-й элемент (нумерация начинается с 0) в контейнере. Различие между ними в том, что a.at(n) выполняет проверку границ и генерирует исключение out_of_range, если n находится вне допустимого диапазона значений контейнера. Кроме того, может вызывать удивление то, что, например, метод push_front() определен для list и deque, но не для vector. Предположим, что требуется вставить новое значение в начало вектора, состоящего из 100 элементов. Чтобы освободить место, нужно переместить 99-й элемент в позицию 100, затем 98-й элемент – в позицию 99 и т.д. Это операция имеет линейную сложность с точки зрения времени выполнения, поскольку перемещение 100 элементов займет в 100 раз больше времени, чем перемещение единственного элемента. Но предполагается, что операции из табл. 16.8 реализованы только в том случае, если они могут быть выполнены за постоянное время. Проектные решения списков и двусторонних очередей, однако, позволяют добавлять элементы в начало без необходимости перемещения других элементов в новые позиции, поэтому они могут реализовать push_front() с постоянным временем выполнения. Выполнение функций push_front() и push_back() проиллюстрировано на рис. 16.4.

Давайте рассмотрим шесть типов контейнеров-последовательностей более подробно.

vector

Вы уже видели несколько примеров, использующих шаблон vector, который объявлен в заголовочном файле vector. Выражаясь кратко, vector – это представление массива в виде класса. Этот класс поддерживает автоматическое управление памятью, которое позволяет динамически менять размер объекта vector, увеличивая и уменьшая его при добавлении или удалении элементов. Он предоставляет произвольный доступ к элементам. Элементы могут добавляться в конец и удаляться с конца за постоянное время, но вставка и удаление в начале и середине – операции с линейным временем выполнения.

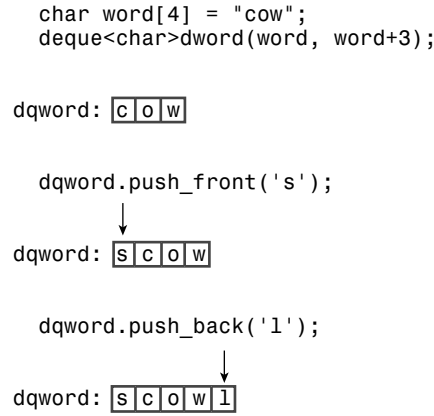


Рис. 16 4. *Функции* `push_front()` *и* `push_back()`

В дополнение к тому, что `vector` является последовательностью, данный контейнер также представляет собой модель концепции *обратимого контейнера*. Это добавляет два метода класса: `rbegin()`, возвращающий итератор на первый элемент обратной последовательности, и `rend()`, возвращающий итератор на элемент, который расположен за последним в обратной последовательности. Таким образом, если `dice` — контейнер типа `vector<int>`, а `Show(int)` — функция, отображающая целое значение, то следующий код отображает содержимое `dice` сначала в прямом, затем в обратном порядке:

```
for_each(dice.begin(), dice.end(), Show);   // отображение в прямом порядке
cout << endl;
for_each(dice.rbegin(), dice.rend(), Show); // отображение в обратном порядке
cout << endl;
```

Итератор, возвращаемый этими двумя методами, относится к типу класса `reverse_iterator`. Вспомните, что операция инкремента, подобная выполняемой итератором, ведет к его перемещению по обратимому контейнеру в обратном порядке.

Класс шаблона `vector` — простейший из типов последовательностей и считается типом, который должен использоваться по умолчанию, если только не выяснится, что какие-то другие типы контейнеров больше удовлетворяют требованиям программы.

deque

Класс шаблона `deque` (объявленный в заголовочном файле `deque`) представляет собой двустороннюю очередь — тип, кратко называемый *дека*. В том виде, каком он реализован в STL, он очень напоминает контейнер `vector`, поддерживая произвольный доступ. Основное различие между ними состоит в том, что вставка и удаление элементов из начала объекта `deque` — операция, выполняемая за постоянное время, в то время как для объекта `vector` эти операции линейны во времени. Поэтому, если большинство операций выполняется в начале и конце последовательности, стоит подумать о применении структуры данных `deque`.

Цель обеспечения постоянного времени вставки и удаления на обоих концах объекта `deque` делает его архитектуру более сложной, чем у `vector`. Таким образом, хотя оба они предоставляют произвольный доступ к элементам, а также линейную во времени вставку и удаление из середины последовательности, контейнер `vector` должен обеспечить более быстрое выполнение этих операций.

list

Класс шаблона list (объявленный в заголовочном файле list) представляет двусвязный список. Каждый его элемент, за исключением первого и последнего, связан с предшествующим и последующим элементом, откуда следует, что такой список можно проходить в обоих направлениях. Принципиальное различие между list и vector заключается в том, что list обеспечивает вставку и удаление за постоянное время в любой позиции списка. (Вспомните, что шаблон vector обеспечивает линейную во времени вставку и удаление, за исключением конца последовательности, где время вставки и удаления постоянно.) Таким образом, vector делает упор на быстрый произвольный доступ, а list — на быструю вставку и удаление элементов.

Подобно vector, класс шаблона list — обратимый контейнер. В отличие от vector, list не поддерживает форму записи массива и произвольный доступ. В отличие от итератора vector, итератор list продолжает указывать на тот же элемент даже после вставки или удаления элементов. Например, пусть имеется итератор, указывающий на пятый элемент контейнера vector. Далее предположим, что вы вставляете элемент в начало контейнера. Для освобождения места все другие элементы должны быть перемещены, поэтому после вставки пятый элемент содержит значение, которое до вставки было четвертым. То есть итератор указывает на ту же позицию, но на другие данные. Однако вставка нового элемента в список не перемещает существующих элементов; она лишь изменяет информацию о связях. Итератор, указывавший на определенный элемент, по-прежнему указывает на него же, но он может быть связан с другими элементами, нежели ранее.

Класс шаблона list имеет ряд ориентированных на списки функций-членов в дополнение к тем, что определены для последовательностей и обратимых контейнеров. Многие из них перечислены в табл. 16.9. (Полный список всех функций и методов STL приведен в приложении Ж.) Обычно о параметре шаблона Alloc можно не беспокоиться, поскольку для него предусмотрено значение по умолчанию.

Таблица 16.9. Некоторые функции-члены класса list

Функция	Описание
`void merge(list<T, Alloc>& x)`	Объединяет список x с вызывающим списком. Оба списка должны быть отсортированы. Результирующий отсортированный список помещается в вызывающий список, а x остается пустым. Эта функция обладает линейной сложностью в плане времени выполнения
`void remove(const T & val)`	Удаляет все экземпляры val из списка. Эта функция обладает линейной сложностью в плане времени выполнения
`void sort()`	Сортирует список, применяя операцию <; время выполнения равно N log N для N элементов
`void splice(iterator pos, list<T, Alloc> x)`	Вставляет содержимое списка x перед позицией pos и оставляет x пустым. Эта функция выполняется за постоянное время
`void unique()`	Сворачивает повторяющиеся соседние элементы в один. Эта функция обладает линейной сложностью в плане времени выполнения

Код в листинге 16.12 иллюстрирует работу этих методов, а также метода insert(), который определен для классов STL, моделирующих последовательности.

Листинг 16.12. list.cpp

```
// list.cpp -- использование списка
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

void outint(int n) {std::cout << n << " ";}

int main()
{
   using namespace std;
   list<int> one(5, 2);                              // список из 5 двоек
   int stuff[5] = {1,2,4,8, 6};
   list<int> two;
   two.insert(two.begin(),stuff, stuff + 5 );
   int more[6] = {6, 4, 2, 4, 6, 5};
   list<int> three(two);
   three.insert(three.end(), more, more + 6);
   cout << "List one: ";                            // первый список
   for_each(one.begin(),one.end(), outint);
   cout << endl << "List two: ";                    // второй список
   for_each(two.begin(), two.end(), outint);
   cout << endl << "List three: ";                  // третий список
   for_each(three.begin(), three.end(), outint);
   three.remove(2);
   cout << endl << "List three minus 2s: ";
                // вычитание второго списка из третьего
   for_each(three.begin(), three.end(), outint);
   three.splice(three.begin(), one);
   cout << endl << "List three after splice: ";     // третий список после splice()
   for_each(three.begin(), three.end(), outint);
   cout << endl << "List one: ";
   for_each(one.begin(), one.end(), outint);
   three.unique();
   cout << endl << "List three after unique: ";     // третий список после unique()
   for_each(three.begin(), three.end(), outint);
   three.sort();
   three.unique();
   cout << endl << "List three after sort & unique: ";
                // третий список после sort() и unique()
   for_each(three.begin(), three.end(), outint);
   two.sort();
   three.merge(two);
   cout << endl << "Sorted two merged into three: ";
                // слияние отсортированного второго списка с третьим
   for_each(three.begin(), three.end(), outint);
   cout << endl;

   return 0;
}
```

Ниже показан вывод программы из листинга 16.12:

```
List one: 2 2 2 2 2
List two: 1 2 4 8 6
List three: 1 2 4 8 6 6 4 2 4 6 5
List three minus 2s: 1 4 8 6 6 4 4 6 5
List three after splice: 2 2 2 2 2 1 4 8 6 6 4 4 6 5
```

```
List one:
List three after unique: 2 1 4 8 6 4 6 5
List three after sort & unique: 1 2 4 5 6 8
Sorted two merged into three: 1 1 2 2 4 4 5 6 6 8 8
```

Замечания по программе

Для отображения списков программа из листинга 16.12 использует алгоритм `for_each()` и функцию `outint()`. В C++11 вместо них можно было бы использовать цикл `for`, основанный на диапазоне:

```
for (auto x : three) cout << x << " ";
```

Основное различие между `insert()` и `splice()` состоит в том, что `insert()` вставляет копию исходного диапазона в место назначения, в то время как `splice()` перемещает исходный диапазон в место назначения. То есть после того, как содержимое `one` сливается с `three`, `one` остается пустым. (Метод `splice()` имеет дополнительные прототипы для перемещения отдельных элементов либо их диапазонов.) Метод `splice()` оставляет итераторы корректными. Таким образом, если определенный итератор установлен для указания на элемент из `one`, то он указывает на тот же элемент и после того, как `splice()` переместит его в `three`.

Обратите внимание, что `unique()` всего лишь удаляет соседние повторяющиеся элементы, оставляя по одному экземпляру. После того, как программа выполнит `three.unique()`, список `three` будет по-прежнему содержать две четверки и две шестерки, которые не были соседними. Но применение `sort()`, а затем `unique()` обеспечит уникальность каждого элемента в списке.

Существует автономная функция `sort()` (см. листинг 16.9), но она требует итераторов произвольного доступа. Поскольку цена быстрой вставки — отказ от произвольного доступа, автономную функцию `sort()` нельзя использовать со списками. Поэтому класс включает в себя версию-член, которая работает в рамках ограничений, накладываемых классом.

Набор инструментов класса `list`

Методы класса `list` образуют удобный набор инструментов. Предположим, например, что требуется организовать два списка почтовой рассылки. Можно было бы отсортировать каждый из списков, объединить их, а затем применить метод `unique()` для удаления повторяющихся записей.

Каждый из методов `sort()`, `merge()` и `unique()` также имеет версию, принимающую дополнительный аргумент, который указывает альтернативную функцию, предназначенную для сравнения элементов. Аналогично, метод `remove()` имеет версию с дополнительным аргументом, который указывает функцию, используемую для определения того, удален ли элемент. Эти аргументы являются примерами функций-предикатов — темы, к которой мы вернемся позднее.

`forward_list` (C++11)

В C++11 появился новый класс контейнера `forward_list`. Этот класс реализует односвязный список. В таком списке каждый элемент связан только со следующим элементом, но не с предыдущим. Поэтому такой класс требует только однонаправленного, а не двунаправленного итератора.

Таким образом, в отличие от `vector` и `list`, `forward_list` не является обратимым контейнером. По сравнению с `list`, `forward_list` проще, компактнее, но предлагает меньше возможностей.

queue

Класс шаблона queue (объявленный в заголовочном файле queue, в прошлом — queue.h) представляет собой класс адаптера. Вспомните, что шаблон ostream_iterator — адаптер, который позволяет выходному потоку использовать интерфейс итератора. Подобным же образом шаблон queue позволяет лежащему в его основе классу (по умолчанию deque) испюльзовать типичный интерфейс очереди.

Класс шаблона queue накладывает больше ограничений, чем deque. Он не только не позволяет произвольный доступ к элементам очереди, но даже не разрешает выполнять итерацию по ее элементам. Взамен queue ограничивается базовыми операциями, определяющими очередь. Можно добавлять элемент в конец очсреди, удалять элемент из ее начала, просматривать значения первого и последнего элементов, проверять количество элементов, а также проверять, не пуста ли очередь. Эти операции перечислены в табл. 16.10.

Следует отметить, что pop() — это метод удаления данных, а не их извлечения. Если требуется использовать значение из очереди, сначала нужно вызвать метод front() для извлечения значения, а затем pop() — для удаления его из очереди.

Таблица 16.10. Операции queue

Метод	Описание
bool empty() const	Возвращает true, если очередь пуста, в противном случае — false
size_type size() const	Возвращает количество элементов в очереди
T& front()	Возвращает ссылку на элемент, находящийся в начале очереди
T& back()	Возвращает ссылку на элемент, находящийся в конце очереди
void push(const T& x)	Вставляет x в конец очереди
void pop()	Удаляет элемент из начала очереди

priority_queue

Класс шаблона priority_queue (объявленный в заголовочном файле queue) представляет собой еще один класс адаптера. Он поддерживает те же операции, что и queue. Главное отличие между этими двумя классами состоит в том, что в priority_queue наибольшее значение перемещается в начало очереди. Внутреннее различие в том, что по умолчанию классом, лежащим в его основе, является vector. Операцию сравнения, используемую для определения того, что должно находиться в голове очереди, можно изменить, передавая необязательный аргумент конструктору:

```
priority_queue<int> pq1;                    // версия по умолчанию
priority_queue<int> pq2(greater<int>);      // использование greater<int>
                                            // для упорядочения
```

Функция greater<>() — это предопределенный функциональный объект, который обсуждается далее в главе.

stack

Подобно queue, stack (объявленный в заголовочном файле stack, ранее известный как stack.h) — это класс адаптера. Он предоставляет лежащему в его основе классу (по умолчанию vector) типичный интерфейс стека.

Класс шаблона stack более ограничен, чем vector. Он не только не разрешает произвольный доступ к элементам стека, но также не позволяет выполнять итерацию по своим элементам. Вместо этого stack ограничивается базовыми операциями, оп-

ределяющими стек. Можно заталкивать значение в вершину стека, выталкивать элемент с вершины, просматривать элемент, находящийся на вершине, запрашивать количество элементов, а также проверять, не пуст ли стек. Эти операции перечислены в табл. 16.11.

Таблица 16.11. Операции класса `stack`

Метод	Описание
`bool empty() const`	Возвращает `true`, если стек пуст, в противном случае — `false`
`size_type size() const`	Возвращает количество элементов в стеке
`T& top()`	Возвращает ссылку на элемент, находящийся на вершине стека
`void push(const T& x)`	Вставляет `x` в вершину стека
`void pop()`	Удаляет элемент из вершины стека

Почти так же, как и с `queue`, если требуется использовать значение из стека, сначала нужно с помощью `top()` извлечь значение, а затем посредством `pop()` удалить его из стека.

`array` (C++11)

Класс шаблона `array`, представленный в главе 4 и определенный в заголовочном файле `array`, не является контейнером STL, поскольку имеет фиксированный размер. Поэтому операции, которые должны были бы изменять размер контейнера, такие как `push_back()` и `insert()`, для класса `array` не определены. Однако определены те функции-члены, которые имеют для него смысл, например, `operator[]()` и `at()`. Кроме того, с объектами массивов можно использовать многие стандартные алгоритмы STL, такие как `copy()` и `for_each()`.

Ассоциативные контейнеры

Ассоциативный контейнер — еще одно уточнение концепции контейнеров. Ассоциативный контейнер связывает значение с ключом, который служит для отыскания значения. Например, значения могут быть структурами, представляющими информацию о сотрудниках, такую как фамилия, адрес, номер офиса, домашний и рабочий телефоны, медицинская карточка и т.д., а ключом может быть уникальный табельный номер сотрудника. Чтобы извлечь информацию о сотруднике, программа должна использовать ключ для обнаружения структуры, описывающей сотрудника. Вспомните, что в общем случае для контейнера `X` выражение `X::value_type` указывает тип значений, хранимых в контейнере. Для ассоциативного контейнера выражение `X::key_type` указывает тип, применяемый для ключей.

Мощь ассоциативных контейнеров в том, что они предоставляют быстрый доступ к своим элементам. Подобно последовательности, ассоциативный контейнер позволяет вставлять элементы; однако нельзя указать определенное местоположение для вставляемых элементов. Это связано с тем, что обычно ассоциативный контейнер обладает конкретным алгоритмом для определения места помещения данных, позволяя быстро извлекать информацию.

Как правило, ассоциативные контейнеры реализуются с помощью той или иной формы дерева. *Дерево* — это структура данных, в которой корневой узел связан с одним или двумя другими узлами, каждый из которых, в свою очередь, связан с одним или двумя узлами, образуя ветвящуюся структуру. Наличие узлов позволяет сравнительно просто добавлять или удалять элементы данных, во многом подобно тому, как это име-

ет место в связных списках. Но по сравнению со списком дерево обеспечивает значительно более быстрый поиск.

Библиотека STL предлагает четыре ассоциативных контейнера: `set`, `multiset`, `map` и `multimap`. Первые два типа объявлены в заголовочном файле `set` (ранее — раздельно в `set.h` и в `multiset.h`), а вторые два типа объявлены в заголовочном файле `map` (в прошлом — раздельно в `map.h` и в `multimap.h`).

Простейшим контейнером из четырех является `set`. Тип его значения совпадает с типом ключа, а ключи уникальны — т.е. в наборе хранится не более одного экземпляра каждого значения ключа. Действительно, для `set` значение элемента является также его ключом. Тип `multiset` подобен `set`, за исключением того, что он может содержать более одного значения с одним и тем же ключом. Например, если типом ключа и значения является `int`, то объект `multiset` может содержать значения 1, 2, 2, 2, 3, 5, 7 и 7.

В случае `map` тип значения отличается от типа ключа, причем ключи уникальны и на каждый ключ приходится только одно значение. Тип `multimap` подобен `map`, за исключением того, что один ключ может быть связан с несколькими значениями.

С этими типами связано слишком много информации, чтобы ее можно было осветить в настоящей главе (но все методы перечислены в приложении Ж), поэтому рассмотрим простые примеры применения `set` и `multimap`.

Пример использования класса `set`

Класс `set` из STL моделирует несколько концепций. Это ассоциативный, обратимый и отсортированный набор с уникальными ключами, поэтому он не может содержать более одного заданного значения. Подобно `vector` и `list`, `set` использует параметр шаблона для указания хранимого типа:

```
set<string> A;                    // набор строковых объектов
```

Необязательный второй аргумент шаблона может служить для указания функции сравнения или объекта, который должен использоваться для упорядочивания ключей. По умолчанию применяется шаблон `less<>` (обсуждаемый ниже). Старые реализации C++ могут не предоставлять значений по умолчанию, а потому требуют явного указания второго параметра шаблона:

```
set<string, less<string> > A;    // старая реализация
```

Рассмотрим следующий код:

```
const int N = 6;
string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
set<string> A(s1, s1 + N);     // инициализация набора A диапазоном массива
ostream_iterator<string, char> out(cout, " ");
copy(A.begin(), A.end(), out);
```

Как и другие контейнеры, `set` имеет конструктор (см. табл. 16.6), который принимает в качестве аргументов диапазон итераторов. Это обеспечивает простой способ инициализации набора содержимым массива. Вспомните, что последний элемент диапазона — это на самом деле элемент, расположенный за последним, а `s1 + N` указывает на одну позицию за концом массива `s1`. Вывод этого фрагмента кода иллюстрирует, что ключи уникальны (строка `"for"` появляется дважды в массиве, но один раз — в наборе), а также то, что набор отсортирован:

```
buffoon can for heavy thinkers
```

В математике определены стандартные операции для наборов (множеств). Например, объединение двух наборов — это набор, состоящий из содержимого этих двух наборов. Если определенное значение — общее для двух наборов, то вследствие уни-

кальности ключей оно появляется в их объединении только один раз. Пересечсние двух наборов — это набор, состоящий из элементов, общих для обоих этих наборов. Разность двух наборов — это первый набор за вычетом элементов, общих для обоих.

Библиотека STL предоставляет алгоритмы, которые поддерживают эти операции. Они представляют собой общие функции, а не методы, поэтому не ограничены объектами set. Однако все объекты set автоматически удовлетворяют обязательному условию применения этих алгоритмов — а именно тому, что контейнер должсн быть отсортирован. Функция set_union() принимает пять итераторов в качестве аргументов. Первые два определяют диапазон одного набора, вторые два — диапазон второго набора, а последний — это выходной итератор, указывающий местоположение, куда следует копировать результирующий набор. Например, для вывода объединения наборов A и B можно воспользоваться показанным ниже оператором:

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
        ostream_iterator<string, char> out(cout, " "));
```

Предположим, что результат должен быть помещен в набор C, а не отображен на экране. В этом случае в последнем аргументе понадобится передать итератор, указывающий на набор C. Очевидным выбором представляется C.begin(), однако это не работает по двум причинам. Первая причина в том, что ассоциативные наборы интерпретируют ключи как постоянные значения, поэтому итератор, возвращенный C.begin(), будет итератором-константой, который не может использоваться в качестве выходного итератора. Вторая причина невозможности непосредственного применения C.begin() заключается в том, что set_union(), подобно copy(), перезаписывает существующие данные в контейнере и требует, чтобы в контейнере было достаточно места для хранения новой информации. Набор C, будучи пустым, не удовлетворяет этому требованию. Но рассмотренный ранее шаблон insert_iterator решает обе проблемы. Ранее было показано, что он превращает копирование во вставку. Он также моделирует концепцию выходного итератора, поэтому его можно использовать для выполнения записи в контейнер. Таким образом, можно создать анонимный insert_iterator для копирования информации в набор C. Конструктор, как вы должны помнить, принимает в качестве аргументов имя контейнера и итератор:

```
set_union(A.begin(), A.end(), B.begin(), B.end(),
        insert_iterator<set<string> >(C, C.begin()));
```

Функции set_intersection() и set_difference() находят пересечение и разность двух наборов и обладают тем же интерфейсом, что и set_union().

Два удобных метода set — это lower_boind() и upper_bound(). Метод lower_boind() принимает значение типа ключа в качестве аргумента и возвращаст итератор, указывающий на первый член набора, который не меньше ключевого аргумента. Аналогично, upper_bound() принимает ключ в качестве аргумента и возвращает итератор, указывающий на первый член набора, который больше ключевого аргумента. Например, если имеется набор строк, эти методы можно применять для определсния диапазона, включающего все строки в наборе от "b" до "f".

Поскольку сортировка определяет, куда будут помещаться добавления к набору, класс имеет методы, которые лишь указывают добавляемые данные без указания позиции. Например, если A и B — наборы строк, можно использовать следующий код:

```
string s("tennis");
A.insert(s);                        // вставка значения
B.insert(A.begin(), A.end());       // вставка диапазона значений
```

Эти применения наборов иллюстрируются в листинге 16.13.

Листинг 16.13. setops.cpp

```
// setops.cpp -- некоторые операции с наборами
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <iterator>

int main()
{
   using namespace std;
   const int N = 6;
   string s1[N] = {"buffoon", "thinkers", "for", "heavy", "can", "for"};
   string s2[N] = {"metal", "any", "food", "elegant", "deliver","for"};

   set<string> A(s1, s1 + N);
   set<string> B(s2, s2 + N);

   ostream_iterator<string, char> out(cout, " ");
   cout << "Set A: ";                          // набор A
   copy(A.begin(), A.end(), out);
   cout << endl;
   cout << "Set B: ";                          // набор B
   copy(B.begin(), B.end(), out);
   cout << endl;

   cout << "Union of A and B:\n";              // объединение A и B
   set_union(A.begin(), A.end(), B.begin(), B.end(), out);
   cout << endl;

   cout << "Intersection of A and B:\n";       // пересечение A и B
   set_intersection(A.begin(), A.end(), B.begin(), B.end(), out);
   cout << endl;

   cout << "Difference of A and B:\n";         // разность A и B
   set_difference(A.begin(), A.end(), B.begin(), B.end(), out);
   cout << endl;

   set<string> C;
   cout << "Set C:\n";                         // набор C
   set_union(A.begin(), A.end(), B.begin(), B.end(),
      insert_iterator<set<string> >(C, C.begin()));
   copy(C.begin(), C.end(), out);
   cout << endl;

   string s3("grungy");
   C.insert(s3);
   cout << "Set C after insertion:\n";         // набор C после вставки
   copy(C.begin(), C.end(),out);
   cout << endl;

   cout << "Showing a range:\n";               // вывод диапазона
   copy(C.lower_bound("ghost"),C.upper_bound("spook"), out);
   cout << endl;

   return 0;
}
```

Вывод программы из листинга 16.13 имеет следующий вид:

```
Set A: buffoon can for heavy thinkers
Set B: any deliver elegant food for metal
Union of A and B:
```

```
any buffoon can deliver elegant food for heavy metal thinkers
Intersection of A and B:
for
Difference of A and B:
buffoon can heavy thinkers
Set C:
any buffoon can deliver elegant food for heavy metal thinkers
Set C after insertion:
any buffoon can deliver elegant food for grungy heavy metal thinkers
Showing a range:
grungy heavy metal
```

Подобно большинству примеров в настоящей главе, код в листинге 16.13 применяет "ленивый" способ объявления пространства имен std:

```
using namespace std;
```

Это делается с целью упрощения представления. В приведенных примерах используется настолько много элементов из пространства имен std, что применение директив или операций разрешения контекста придало бы коду несколько громоздкий вид:

```
std::set<std::string> B(s2, s2 + N);
std::ostream_iterator<std::string, char> out(std::cout, " ");
std::cout << "Set A: ";
std::copy(A.begin(), A.end(), out);
```

Пример использования multimap

Как и set, multimap — это обратимый, отсортированный ассоциативный контейнер. Однако при использовании multimap тип ключа отличается от типа значения, а объект multimap может содержать более одного значения, связанного с конкретным ключом.

Простейшее объявление multimap задает тип ключа и тип значения, сохраненные в виде аргументов шаблона. Например, следующее объявление создает объект multimap, который использует int как тип ключа и string — в качестве типа сохраненного значения:

```
multimap<int, string> codes;
```

Необязательный третий аргумент шаблона может применяться для указания функции сравнения или объекта, который будет использоваться для упорядочивания ключа. По умолчанию применяется шаблон less<> (рассмотренный ниже) с типом ключа в качестве параметра. Более старые реализации C++ могут требовать явного указания этого параметра шаблона.

Короче говоря, действительный тип значения объединяет тип ключа и тип данных в единую пару. Для этого STL использует класс шаблона pair<class T, class U> для хранения двух видов значений в одном объекте. Если keytype — тип ключа, а datatype — тип сохраненных данных, то типом значения будет pair<const keytype, datatype>. Например, типом значения ранее объявленного объекта codes является pair<const int, string>.

Предположим, что требуется хранить названия городов с кодом региона в качестве ключа. Это вполне соответствует объявлению объекта codes, которое использует int как тип ключа и string — как тип данных. Один из возможных подходов — создание пары и вставка ее в объект multimap:

```
pair<const int, string> item(213, "Los Angeles");
codes.insert(item);
```

Либо можно создать анонимный объект pair и вставить его в единственном операторе:

```
codes.insert(pair<const int, string> (213, "Los Angeles"));
```

Поскольку элементы сортируются по ключу, нет необходимости указывать позицию вставки.

Располагая объектом pair, получать доступ к двум его компонентам можно с помощью членов first и second:

```
pair<const int, string> item(213, "Los Angeles");
cout << item.first << ' ' << item.second << endl;
```

А как насчет получения информации об объекте multimap? Функция-член count() принимает в качестве аргумента ключ и возвращает количество элементов, имеющих такое значение ключа. Функции-члены lower_bound() и upper_bound() принимают ключ и работают так же, как в классе set. Функция-член equal_range() также принимает в качестве аргумента ключ и возвращает итераторы, представляющие диапазон значений, соответствующих этому ключу. Чтобы вернуть два значения, метод упаковывает их в один объект pair, на этот раз с обоими аргументами шаблона — итераторами. Например, следующий код выведет список городов из объекта codes, у которых код региона равен 718:

```
pair<multimap<KeyType, string>::iterator,
   multimap<KeyType, string>::iterator> range
                  = codes.equal_range(718);
cout << "Cities with area code 718:\n";
std::multimap<KeyType, std::string>::iterator it;
for (it = range.first; it != range.second; ++it)
   cout << (*it).second << endl;
```

Объявления, подобные приведенному выше, способствовали появлению средства автоматического выведения типа C++11, которое позволяет упростить код следующим образом:

```
auto range = codes.equal_range(718);
cout << "Cities with area code 718:\n";
for (auto it = range.first; it != range.second; ++it)
   cout << (*it).second << endl;
```

В листинге 16.14 демонстрируется применение большей части описанных методик. В нем также используется typedef для упрощения написания кода.

Листинг 16.14. multimap.cpp

```
// multimap.cpp -- использование multimap
#include <iostream>
#include <string>
#include <map>
#include <algorithm>

typedef int KeyType;
typedef std::pair<const KeyType, std::string> Pair;
typedef std::multimap<KeyType, std::string> MapCode;

int main()
{
   using namespace std;
   MapCode codes;
```

```
    codes.insert(Pair(415, "San Francisco"));
    codes.insert(Pair(510, "Oakland"));
    codes.insert(Pair(718, "Brooklyn"));
    codes.insert(Pair(718, "Staten Island"));
    codes.insert(Pair(415, "San Rafael"));
    codes.insert(Pair(510, "Berkeley"));

    cout << "Number of cities with area code 415: "
         << codes.count(415) << endl;     // количество городов с кодом региона 415
    cout << "Number of cities with area code 718: "
         << codes.count(718) << endl;     // количество городов с кодом региона 718
    cout << "Number of cities with area code 510: "
         << codes.count(510) << endl;     // количество городов с кодом региона 510
    cout << "Area Code City\n";
    MapCode::iterator it;

    for (it = codes.begin(); it != codes.end(); ++it)
       cout << " " << (*it).first << " "
            << (*it).second << endl;

    pair<MapCode::iterator, MapCode::iterator> range
       = codes.equal_range(718);
    cout << "Cities with area code 718:\n";
    for (it = range.first; it != range.second; ++it)
       cout << (*it).second << endl;

    return 0;
}
```

Вывод программы из листинга 16.14 имеет следующий вид:

```
Number of cities with area code 415: 2
Number of cities with area code 718: 2
Number of cities with area code 510: 2
Area Code  City
   415    San Francisco
   415    San Rafael
   510    Oakland
   510    Berkeley
   718    Brooklyn
   718    Staten Island
Cities with area code 718:
Brooklyn
Staten Island
```

Неупорядоченные ассоциативные контейнеры (C++11)

Неупорядоченный ассоциативный контейнер — еще одно уточнение концепции контейнеров. Подобно ассоциативному контейнеру, неупорядоченный ассоциативный контейнер связывает значение с ключом и использует ключ для отыскания значения. Принципиальное различие между ними в том, что в основе ассоциативных контейнеров лежат древовидные структуры, тогда как неупорядоченные ассоциативные контейнеры построены на основе другой формы структур данных, называемой *хеш-таблицей*. Цель состоит в получении контейнеров, в которых добавление и удаление элементов осуществляется сравнительно быстро и для которых существуют эффективные алгоритмы поиска. Доступны четыре типа неупорядоченных ассоциативных контейнеров: `unordered_set`, `unordered_multiset`, `unordered_map` и `unordered_multimap`. Несколько подробнее эти дополнения описаны в приложении Ж.

Функциональные объекты (функторы)

Многие алгоритмы STL используют *функциональные объекты*, которые также называются *функторами*. *Функтор* — это любой объект, который может использоваться с операцией `()` в манере, подобной функции. Это включает нормальные имена функций, указатели на функции и объекты классов, с перегруженной операцией `()` — т.е. классы, для которых определена несколько необычно выглядящая функция `operator()()`. Например, можно было бы определить такой класс:

```
class Linear
{
private:
   double slope;
   double y0;
public:
   Linear(double sl_ = 1, double y_ = 0)
      : slope(sl_), y0(y_) {}
   double operator()(double x) {return y0 + slope * x; }
};
```

Тогда перегруженная операция `()` позволит использовать объекты `Linear` подобно функциям:

```
Linear f1;
Linear f2(2.5, 10.0);
double y1 = f1(12.5);  // правой частью является f1.operator()(12.5)
double y2 = f2(0.4);
```

В этом примере `y1` вычисляется с помощью выражения `0 + 1 * 12.5`, а `y2` — с помощью выражения `10.0 + 2.5 * 0.4`. Значения `y0` и `slope` в выражении `y0 + slope * x` поступают из конструктора объекта, а значение `x` — из аргумента `operator()()`.

Помните функцию `for_each`? Она применяла указанную функцию к каждому члену из заданного диапазона:

```
for_each(books.begin(), books.end(), ShowReview);
```

В общем случае третий аргумент может быть функтором, а не обычной функцией. В действительности возникает вопрос: как объявить третий аргумент? Его нельзя объявлять как указатель на функцию, поскольку указатель на функцию подразумевает наличие типов аргументов. Так как контейнер может содержать практически любой тип, заранее не известно, какой конкретный тип аргумента должен быть использован. Библиотека STL решает эту проблему, применяя шаблоны. Прототип `for_each` выглядит следующим образом:

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

Ниже показан прототип `ShowReview()`:

```
void ShowReview(const Review &);
```

В результате типом идентификатора `ShowReview` становится `void (*)(const Review &)`, поэтому именно данный тип назначается аргументу шаблона `Function`. При других вызовах функций аргумент `Function` мог бы представлять тип класса с перегруженной операцией `()`. В любом случае код `for_each()` будет содержать выражение, использующее конструкцию `f()`. В примере с `ShowReview()` идентификатор `f` — указатель на функцию, а конструкция `f()` вызывает функцию. Если последний аргумент функции `for_each()` — объект, то `f()` становится объектом, который вызывает свою перегруженную операцию `()`.

Концепции функторов

Подобно тому, как библиотека STL определяет концепции контейнеров и итераторов, она определяет также концепции функторов:

- *генератор* – это функтор, который может быть вызван без аргументов;
- *унарная функция* – функтор, который может быть вызван с одним аргументом;
- *бинарная функция* – функтор, который может быть вызван с двумя аргументами.

Например, функтор, поддерживающий `for_each()`, должен быть унарной функцией, поскольку он применяется к одному элементу контейнера за раз.

Конечно, приведенные концепции имеют уточнения:

- унарная функция, которая возвращает булевское значение, представляет собой *предикат*;
- бинарная функция, которая возвращает булевское значение, представляет собой *бинарный предикат*.

Некоторые функции STL требуют предикатов либо бинарных предикатов в качестве аргументов. Например, в листинге 16.9 используется версия `sort()`, которая в третьем аргументе принимает бинарный предикат:

```
bool WorseThan(const Review & r1, const Review & r2);
...
sort(books.begin(), books.end(), WorseThan);
```

Шаблон `list` имеет функцию-член `remove_if()`, которая принимает предикат в качестве аргумента. Она применяет предикат к каждому члену указанного диапазона, удаляя те элементы, для которых предикат возвращает значение `true`. Например, следующий код удалил бы из списка `three` все элементы, которые больше `100`:

```
bool tooBig(int n){ return n > 100; }
list<int> scores;
...
scores.remove_if(tooBig);
```

Кстати, последний пример показывает, где могут быть полезны классы функторов. Предположим, например, что из второго списка требуется удалить все значения, которые больше `200`. Было бы неплохо, если бы функции `tooBig()` можно было передать граничное значение, чтобы использовать ее с различными значениями, но предикат принимает только один аргумент. Однако если вы проектируете класс `TooBig`, то для передачи дополнительной информации вместо аргументов функции можно применять члены класса:

```
template<class T>
class TooBig
{
private:
   T cutoff;
public:
   TooBig(const T & t) : cutoff(t) {}
   bool operator()(const T & v) { return v > cutoff; }
};
```

Здесь одно значение (`v`) передается как аргумент функции, а второй аргумент (`cutoff`) устанавливается конструктором класса. Располагая таким определением,

можно инициализировать разные объекты TooBig разными граничными значениями для их использования в вызовах remove_if(). Этот подход продемонстрирован в листинге 16.15.

Листинг 16.15. functor.cpp

```
// functor.cpp — использование класса functor
#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

template<class T>  // класс функтора определяет operator()()
class TooBig
{
private:
   T cutoff;
public:
   TooBig(const T & t) : cutoff(t) {}
   bool operator()(const T & v) { return v > cutoff; }
};

void outint(int n) {std::cout << n << " ";}

int main()
{
   using std::list;
   using std::cout;
   using std::endl;

   TooBig<int> f100(100);                        // предельное значение = 100
   int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
   list<int> yadayada(vals, vals + 10);          // конструктор диапазона
   list<int> etcetera(vals, vals + 10);

   // Вместо этого в C++11 можно использовать следующий код:
   // list<int> yadayada = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
   // list<int> etcetera {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
   cout << "Original lists:\n";                  // исходные списки
   for_each(yadayada.begin(), yadayada.end(), outint);
   cout << endl;

   for_each(etcetera.begin(), etcetera.end(), outint);
   cout << endl;
   yadayada.remove_if(f100);                     // использование именованного
                                                 // функционального объекта
   etcetera.remove_if(TooBig<int>(200));         // конструирование
                                                 // функционального объекта
   cout <<"Trimmed lists:\n";                    // усеченные списки

   for_each(yadayada.begin(), yadayada.end(), outint);
   cout << endl;

   for_each(etcetera.begin(), etcetera.end(), outint);
   cout << endl;
   return 0;
}
```

Один из функторов (f100) является объявленным объектом, а второй (TooBig<int>(200)) — анонимным объектом, который создан вызовом конструктора. Вот как выглядит вывод программы из листинга 16.15:

```
Original lists:
50 100 90 180 60 210 415 88 188 201
50 100 90 180 60 210 415 88 188 201
Trimmed lists:
50 100 90 60 88
50 100 90 180 60 88 188
```

Предположим, что имеется шаблонная функция с двумя аргументами:

```
template <class T>
bool tooBig(const T & val, const T & lim)
{
   return val > lim;
}
```

Чтобы преобразовать ее в функциональный объект с одним аргументом, можно использовать класс:

```
template<class T>
class TooBig2
{
private:
   T cutoff;
public:
   TooBig2(const T & t) : cutoff(t) {}
   bool operator()(const T & v) { return tooBig<T>(v, cutoff); }
};
```

Это значит, что можно написать следующий код:

```
TooBig2<int> tB100(100);
int x;
cin >> x;
if (tB100(x))  // то же что и if (tooBig(x,100))
   ...
```

Таким образом, вызов `tB100(100)` эквивалентен `tooBig(x,100)`, но функция с двумя аргументами преобразуется в функциональный объект с одним аргументом, а второй аргумент служит для конструирования функционального объекта. Короче говоря, функтор `TooBig2` — это адаптер функции, приспосабливающий функцию к требованиям другого интерфейса.

Как отмечено в листинге, средство списка инициализаторов C++11 упрощает инициализацию. Код

```
int vals[10] = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
list<int> yadayada(vals, vals + 10);  // конструктор диапазона
list<int> etcetera(vals, vals + 10);
```

можно заменить следующим:

```
list<int> yadayada = {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
list<int> etcetera {50, 100, 90, 180, 60, 210, 415, 88, 188, 201};
```

Предопределенные функторы

Библиотека STL определяет несколько элементарных функторов. Они выполняют такие действия, как сложение двух значений и проверка двух значений на предмет равенства. Функторы предназначены для оказания поддержки тем функциям STL, которые принимают функции в качестве аргументов. Для примера рассмотрим функцию `transform()`. Она имеет две версии. Первая принимает четыре аргумента. Из них

первые два — итераторы, которые задают диапазон контейнера. (Теперь этот подход уже должен быть хорошо знаком.) Третий аргумент — это итератор, указывающий место помещения копии результата. И последний — функтор, который применяется к каждому элементу диапазона для создания каждого нового результирующего элемента. Рассмотрим следующий пример:

```
const int LIM = 5;
double arr1[LIM] = {36, 39, 42, 45, 48};
vector<double> gr8(arr1, arr1 + LIM);
ostream_iterator<double, char> out(cout, " ");
transform(gr8.begin(), gr8.end(), out, sqrt);
```

Этот код вычисляет квадратный корень каждого элемента и отправляет результирующее значение в выходной поток. Итератор места назначения может указывать на исходный диапазон.

Например, замена в этом примере аргумента `out` аргументом `gr8.begin()` привела бы к копированию новых значений поверх старых. Понятно, что применяемый функтор должен быть таким, который работает с одним аргументом.

Вторая версия использует функцию, которая принимает два аргумента, применяя функцию к одному элементу из каждого их двух диапазонов. Она принимает дополнительный аргумент, который является третьим по порядку и идентифицирует начало второго диапазона. Например, если бы `m8` был вторым объектом `vector<double>` и функция `mean(double, double)` возвращала среднее значение двух значений, то следующий фрагмент кода выводил бы среднее значение каждой пары значений из `gr8` и `m8`:

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, mean);
```

Теперь предположим, что требуется сложить два массива. Операцию + использовать в качестве аргумента нельзя, потому что для типа `double` она является встроенной операцией, а не функцией. Можно было бы определить функцию для сложения двух чиссл и воспользоваться ею:

```
double add(double x, double y) { return x + y; }
...
transform(gr8.begin(), gr8.end(), m8.begin(), out, add);
```

Но тогда пришлось бы определять отдельную функцию для каждого типа. Лучше определить шаблон, за исключением тех случаев, когда библиотека STL уже содержит его. Заголовочный файл `functional` (ранее — `function.h`) определяет несколько функциональных объектов класса шаблона, в том числе `plus<>()`.

Применение класса `plus<>` для простого сложения возможно, хотя и неудобно:

```
#include <functional>
...
plus<double> add;              // создание объекта plus<double>
double y = add(2.2, 3.4);      // использование plus<double>::operator()()
```

Однако проще предоставить функциональный объект в виде аргумента:

```
transform(gr8.begin(), gr8.end(), m8.begin(), out, plus<double>() );
```

В этом примере вместо создания именованного объекта с помощью конструктора `plus<double>` создается функтор для выполнения сложения. (Круглые скобки указывают вызов конструктора по умолчанию, который передает сконструированный функциональный объект функции `transform()`.)

Библиотека STL предлагает эквивалентные функторы для всех встроенных арифметических, реляционных и логических операций. Имена этих эквивалентов-функторов перечислены в табл. 16.12. Их можно использовать со встроенными типами C++ или любым определенным пользователем типом, который перегружает соответствующую операцию.

Таблица 16.12. Операции и их эквиваленты–функторы

Операция	Эквивалент-функтор
+	plus
-	minus
*	multiplies
/	divides
%	modulus
-	negate
==	equal_to
!=	not_equal_to
>	greater
<	less
>=	greater_equal
<=	less_equal
&&	logical_and
||	logical_or
!	logical_not

Внимание!

В старых реализациях C++ вместо имени функтора `multiplies` используется `times`.

Адаптируемые функторы и функциональные адаптеры

Все приведенные в табл. 16.12 предопределенные функторы являются *адаптируемыми*. Фактически библиотека STL использует пять связанных концепций: адаптируемые генераторы, адаптируемые унарные функции, адаптируемые бинарные функции, адаптируемые предикаты и адаптируемые бинарные предикаты.

Адаптируемым функтор делает тот факт, что он использует члены `typedef`, указывающие типы их аргументов и тип возвращаемых значений. Эти члены называются `result_type` (тип результата), `first_argument_type` (тип первого аргумента) и `second_argument_type` (тип второго аргумента) и представляют именно то, что подразумевают их имена. Например, возвращаемый тип объекта `plus<int>` указан как `plus<int>::result_type`, и он должен служить `typedef` для `int`.

Важность адаптируемости функтора в том, что тогда он может применяться объектами функционального адаптера, который исходит из существования этих членов `typedef`. Например, функция с аргументом, который является адаптируемым функтором, может использовать член `result_type` для объявления переменной, соответствующей возвращаемому типу функции.

Действительно, STL предоставляет классы функциональных адаптеров, использующих эти средства. Например, предположим, что каждый элемент вектора `gr8` нужно

умножить на 2.5. В этом случае можно применить версию transform() с аргументом – унарной функцией, которая подобна показанной в примере ранее:

```
transform(gr8.begin(), gr8.end(), out, sqrt);
```

Функтор multiplies() может выполнять умножение, но это – бинарная функция. Поэтому необходим адаптер функции, который преобразует функтор с двумя аргументами в функтор с одним аргументом. Приведенный ранее пример с TooBig2 демонстрирует один способ сделать это, но STL автоматизирует этот процесс с помощью классов binder1st и binder2nd, которые преобразуют адаптируемые бинарные функции в адаптируемые унарные функции.

Рассмотрим класс binder1st. Предположим, что имеется адаптируемый бинарный функциональный объект f2(). Можно создать объект binder1st, который привязывает определенное значение по имени val, чтобы оно использовалось в качестве первого аргумента f2():

```
binder1st(f2, val) f1;
```

Тогда вызов f1(x) с его единственным аргументом возвратит то же значение, что и f2() с val в качестве его первого аргумента и аргументом функции f1() в качестве второго аргумента. То есть f1(x) эквивалентна f2(val, x), за исключением того, что это унарная, а не бинарная функция. Иными словами, функция f2() адаптирована. Повторимся еще раз: подобное возможно, только если f2() – адаптируемая функция.

Этот подход может показаться несколько странным. Однако STL предлагает функцию bind1st() для упрощения работы с классом binder1st. Ей передается имя функции и значение, используемое для создания объекта binder1st, а она возвращает объект этого типа. Например, бинарную функцию multiplies() можно преобразовать в унарную, которая умножает свой аргумент на 2.5. Для этого достаточно написать такой оператор:

```
bind1st(multiplies<double>(), 2.5)
```

Таким образом, решение по умножению каждого элемента gr8 на 2.5 и отображению результатов имеет следующий вид:

```
transform(gr8.begin(), gr8.end(), out,
      bind1st(multiplies<double>(), 2.5));
```

Класс binder2nd аналогичен рассмотренному, за исключением того, что он присваивает константу второму аргументу вместо первого. Он имеет вспомогательную функцию bind2nd, которая работает аналогично bind1st.

В листинге 16.16 приведена короткая программа, в которой собраны вместе некоторые из последних рассмотренных примеров.

Листинг 16.16. fundap.cpp

```
// fundap.cpp -- использование адаптеров функций
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>

void Show(double);
const int LIM = 6;
int main()
{
   using namespace std;
```

```
    double arr1[LIM] = {28, 29, 30, 35, 38, 59};
    double arr2[LIM] = {63, 65, 69, 75, 80, 99};
    vector<double> gr8(arr1, arr1 + LIM);
    vector<double> m8(arr2, arr2 + LIM);
    cout.setf(ios_base::fixed);
    cout.precision(1);
    cout << "gr8:\t";
    for_each(gr8.begin(), gr8.end(), Show);
    cout << endl;
    cout << "m8: \t";
    for_each(m8.begin(), m8.end(), Show);
    cout << endl;

    vector<double> sum(LIM);
    transform(gr8.begin(), gr8.end(), m8.begin(), sum.begin(),
          plus<double>());
    cout << "sum:\t";
    for_each(sum.begin(), sum.end(), Show);
    cout << endl;

    vector<double> prod(LIM);
    transform(gr8.begin(), gr8.end(), prod.begin(),
          bind1st(multiplies<double>(), 2.5));
    cout << "prod:\t";
    for_each(prod.begin(), prod.end(), Show);
    cout << endl;

    return 0;
}

void Show(double v)
{
    std::cout.width(6);
    std::cout << v << ' ';
}
```

Вывод программы, приведенной в листинге 16.16, имеет следующий вид:

```
gr8:     28.0   29.0   30.0   35.0   38.0   59.0
m8:      63.0   65.0   69.0   75.0   80.0   99.0
sum:     91.0   94.0   99.0  110.0  118.0  158.0
prod:    70.0   72.5   75.0   87.5   95.0  147.5
```

C++11 предоставляет альтернативу функциональным указателям и функторам — *лямбда-выражения*, которые более подробно рассматриваются в главе 18.

Алгоритмы

Библиотека STL включает в себя множество автономных функций для работы с контейнерами. Некоторые из них уже встречались: `sort()`, `copy()`, `find()`, `for_each()`, `random_shuffle()`, `set_union()`, `set_intersection()`, `set_difference()` и `transform()`. Возможно, вы заметили, что все они характеризуются одинаковым общим подходом — применением итераторов для идентификации диапазонов обрабатываемых данных и места помещения результатов. Некоторые также принимают аргумент — функциональный объект, который используется в процессе обработки данных.

Существуют два основных общих проектных компонента функций алгоритмов. Во-первых, они применяют шаблоны для предоставления обобщенных типов. Во-вторых, они используют итераторы для обеспечения обобщенного представления доступа к

данным в контейнере. Таким образом, функция copy() может работать с контейнером, который содержит значения типа double в массиве, с контейнером, содержащим значения string в связном списке, либо с контейнером, который хранит определенные пользователем объекты в древовидной структуре, как это делает set. Поскольку указатели — это специальный случай итераторов, функции STL, подобные copy(), могут применяться с обычными массивами.

Унифицированное проектное решение контейнеров позволяет реализовать имеющие смысл отношения между контейнерами разных видов. Например, функцию copy() можно использовать для копирования значений обычного массива в объект vector, из объекта vector в объект list и из объекта list в объект set. Операцию == можно применять для сравнения разных видов контейнеров — например, deque и vector. Это возможно потому, что перегруженная операция == для контейнеров применяет итераторы для сравнения содержимого, и в результате объекты deque и vector считаются эквивалентными, если они имеют одно и то же содержимое в одном и том же порядке.

Группы алгоритмов

STL разделяет библиотеку алгоритмов на четыре группы:

- немодифицирующие последовательные операции;
- модифицирующие последовательные операции;
- сортирующие и связанные с ними операции;
- обобщенные числовые операции.

Первые три группы описаны в заголовочном файле algorithm (бывший algo.h), а четвертая группа, будучи специально ориентированной на числовые данные, имеет собственный заголовочный файл numeric (раньше они также были в algo.h).

Немодифицирующие последовательные операции обрабатывают каждый элемент в диапазоне. Эти операции оставляют контейнер без изменений. Например, find() и for_each() относятся к этой категории.

Модифицирующие последовательные операции также обрабатывают каждый элемент в контейнере. Однако, как следует из их названия, они могут изменить содержимое контейнера. Изменения могут касаться значений либо порядка, в котором они сохранены. Функции transform(), random_shuffle() и copy() относятся к этой категории.

Сортирующие и связанные с ними операции включают некоторые сортирующие функции (в том числе sort()), а также ряд других функций, включая операции с наборами (множествами).

Числовые операции включают функции для суммирования содержимого диапазона, вычисления внутреннего произведения двух контейнеров, подсчета частичных сумм и вычисления разностей соседних элементов. Как правило, эти операции характерны для массивов, поэтому vector — это контейнер, который наиболее вероятно будет использован с ними.

Полный перечень этих функций приведен в приложении Ж.

Основные свойства алгоритмов

Как неоднократно было показано в этой главе, функции STL работают с итераторами и диапазонами итераторов. Прототипы функций указывают предположения, сделанные относительно итераторов.

Например, функция copy() имеет следующий прототип:

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

Поскольку идентификаторы InputIterator и OutputIterator — это параметры шаблона, ими с тем же успехом могли быть T и U. Однако в документации по STL для указания концепций, которые эти параметры моделируют, используются имена параметров шаблонов. Поэтому это объявление указывает, что параметрами диапазона должны быть входные итераторы или более мощные итераторы, и что итератором, указывающим место помещения результата, должен быть выходной или более мощный итератор.

Один из способов классификации алгоритмов основан на том, куда должен быть помещен результат работы алгоритма. Некоторые алгоритмы делают свою работу на месте, другие создают копии. Например, по завершении работы функции sort() результат занимает то же местоположение, которое занимали исходные данные. Поэтому sort() — *алгоритм "по месту"*. Однако функция copy() отправляет результат своей работы в другое местоположение, поэтому она является *копирующим алгоритмом*. Функция transform() может делать и то, и другое. Подобно copy(), она использует выходной итератор для указания места помещения результата. Но, в отличие от copy(), transform() позволяет выходному итератору указывать позицию внутри входного диапазона, поэтому она может копировать трансформированные значения поверх исходных.

Некоторые алгоритмы имеют две версии: "по месту" и копирующую. В соответствии с соглашением STL, к имени копирующей версии добавляется _copy. Последняя версия принимает дополнительный параметр — выходной итератор для указания места копирования результата. Например, существует функция replace() с таким прототипом:

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
             const T& old_value, const T& new_value);
```

Она заменяет экземпляр old_value экземпляром new_value. Это происходит на месте. Поскольку данный алгоритм выполняет и чтение, и запись элементов контейнера, типом итератора должен быть ForwardIterator либо более мощный.

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
             OutputIterator result,
             const T& old_value, const T& new_value);
```

На этот раз результирующие данные копируются в новое место, заданное result, поэтому для указания диапазона вполне достаточно входного итератора, выполняющего только чтение.

Обратите внимание, что возвращаемым типом функции replace_copy() является OutputIterator. В соответствии с принятым соглашением, копирующие алгоритмы возвращают итератор, который указывает на позицию, следующую за последним скопированным значением.

Еще одна часто используемая разновидность — функции, имеющие версию, которая выполняет действие условно, в зависимости от результата применения функции к элементу контейнера. Как правило, к именам этих версий добавляется _if. Например, функция replace_if() заменяет старое значение новым, если применение функции к старому значению возвращает true.

Вот ее прототип:

```
template<class ForwardIterator, class Predicate class T>
void replace_if(ForwardIterator first, ForwardIterator last,
                Predicate pred, const T& new_value);
```

(Вспомните, что предикат – это унарная функция, возвращающая значение `bool`.)

Существует также версия по имени `replace_copy_if()`. Догадаться, что она делает, и как выглядит ее прототип, довольно легко. Как и `InputIterator`, `Predicate` представляет собой имя параметра шаблона, которым с равным успехом могло бы быть `T` или `U`. Однако STL выбирает имя `Predicate`, чтобы напомнить пользователю, что действительный аргумент должен быть моделью концепции `Predicate`. Аналогично, в STL применяются термины вроде `Generator` и `BinaryPredicate` для идентификации алгоритмов, которые должны моделировать концепции других функциональных объектов. Имейте в виду, что хотя документация может напомнить требования к итератору или функтору, эти имена – не то, что может проверить компилятор. Использование неподходящего вида итератора может привести к длинному списку сообщений об ошибках, поскольку компилятор будет пытаться создать экземпляр шаблона.

Библиотека STL и класс `string`

Класс `string`, хотя и не является частью библиотеки STL, спроектирован с учетом ее наличия. Так, например, он имеет функции-члены `begin()`, `end()`, `rbegin()` и `rend()`. Это значит, что он может работать с интерфейсом STL.

Код в листинге 16.17 использует STL для отображения всех возможных перестановок букв слова. *Перестановка* – это изменение порядка следования элементов в контейнере. Алгоритм `next_permutation()` преобразует содержимое диапазона в следующую перестановку; в случае строки перестановки выполняются в возрастающем алфавитном порядке. Алгоритм возвращает `true` при успешном продолжении и `false` – в ситуации, когда порядок следования элементов в диапазоне является последним из возможных. Чтобы получить все перестановки диапазона, следует начать с элементов, расположенных в самом первом возможном порядке, и с этой целью в программе используется алгоритм `sort()` из библиотеки STL.

Листинг 16.17. `strgstl.cpp`

```
// strgstl.cpp -- применение STL к строке
#include <iostream>
#include <string>
#include <algorithm>
int main()
{
    using namespace std;
    string letters;
    cout << "Enter the letter grouping (quit to quit): "; // ввод группы букв
    while (cin >> letters && letters != "quit")
    {
        cout << "Permutations of " << letters << endl;     // перестановки группы букв
        sort(letters.begin(), letters.end());
        cout << letters << endl;
        while (next_permutation(letters.begin(), letters.end()))
            cout << letters << endl;
        cout << "Enter next sequence (quit to quit): ";    // ввод следующей группы букв
    }
    cout << "Done.\n";
    return 0;
}
```

Ниже приведен пример запуска программы из листинга 16.17:

```
Enter the letter grouping (quit to quit): awl
Permutations of awl
alw
awl
law
lwa
wal
wla
Enter next sequence (quit to quit): all
Permutations of all
all
lal
lla
Enter next sequence (quit to quit): quit
Done.
```

Обратите внимание, что алгоритм `next_permutation()` автоматически обеспечивает генерацию только уникальных перестановок — вот почему для слова *awl* показано больше перестановок, чем для слова *all*, в котором присутствуют повторяющиеся буквы.

Сравнение функций и методов контейнеров

Иногда возникает ситуация, когда требуется произвести выбор между использованием метода STL либо функции STL. Обычно лучшим вариантом является метод. Во-первых, он должен быть лучше оптимизирован для конкретного контейнера. Во-вторых, будучи функцией-членом, он может пользоваться средствами управления памятью класса шаблона и при необходимости изменять размер контейнера.

Предположим, например, что имеется список чисел, и нужно удалить из него все экземпляры определенного значения, скажем, `4`. Если `la` — это объект `list<int>`, можно применить метод `remove()` списка:

```
la.remove(4);  // удаление всех четверок из списка
```

После вызова этого метода все элементы со значением `4` удаляются из списка и размер списка автоматически изменяется.

В STL существует также алгоритм `remove()` (см. приложение Ж). Вместо того чтобы вызываться объектом, он принимает аргументы, задающие диапазон. Поэтому, если `lb` — это объект `list<int>`, то вызов данной функции может выглядеть следующим образом:

```
remove(lb.begin(), lb.end(), 4);
```

Однако, поскольку функция `remove()` не является членом класса, она не может изменить размер списка. Вместо этого она удостоверяется, что все не удаленные элементы располагаются в начале списка, и возвращает итератор, указывающий на новое значение за концом списка. Затем этот итератор можно применять для корректировки размеров списка. Например, метод `erase()` списка можно использовать для удаления диапазона, который описывает более не нужную часть списка. Работа этого процесса продемонстрирована в листинге 16.18.

Листинг 16.18. `listrmv.cpp`

```
// strgstl.cpp -- применение STL к строке
#include <iostream>
#include <list>
```

```
#include <algorithm>
void Show(int);
const int LIM = 10;
int main()
{
   using namespace std;
   int ar[LIM] = {4, 5, 4, 2, 2, 3, 4, 8, 1, 4};
   list<int> la(ar, ar + LIM);
   list<int> lb(la);
   cout << "Original list contents:\n\t";    // вывод содержимого исходного списка
   for_each(la.begin(), la.end(), Show);
   cout << endl;
   la.remove(4);
   cout << "After using the remove() method:\n";
        // список после использования метода remove()
   cout << "la:\t";
   for_each(la.begin(), la.end(), Show);
   cout << endl;
   list<int>::iterator last;
   last = remove(lb.begin(), lb.end(), 4);
   cout << "After using the remove() function:\n";
        // список после использования функции remove()
   cout << "lb:\t";
   for_each(lb.begin(), lb.end(), Show);
   cout << endl;
   lb.erase(last, lb.end());
   cout << "After using the erase() method:\n";
        // список после использования метода erase()
   cout << "lb:\t";
   for_each(lb.begin(), lb.end(), Show);
   cout << endl;
   return 0;
}
void Show(int v)
{
   std::cout << v << ' ';
}
```

Ниже приведен вывод программы из листинга 16.18:

```
Original list contents:
   4 5 4 2 2 3 4 8 1 4
After using the remove() method:
la: 5 2 2 3 8 1
After using the remove() function:
lb: 5 2 2 3 8 1 4 8 1 4
After using the erase() method:
lb: 5 2 2 3 8 1
```

Как видите, метод `remove()` уменьшает размер списка `la` с 10 до 6 элементов. Однако после вызова функции `remove()` список `lb` по-прежнему содержит 10 элементов. Последние четыре элемента освобождаются, потому что каждый из элементов со значением `4` либо с дублированным значением перемещен ближе к началу списка.

Хотя обычно методы подходят лучше, обычные функции более универсальны. Как вы видели, их можно использовать с массивами и объектами `string`, равно как и с контейнерами STL. И их можно применять с контейнерами смешанных типов, например, для сохранения данных контейнера `vector` в списке или наборе.

Использование STL

STL — это библиотека, отдельные части которой предназначены для совместной работы. Компоненты STL являются инструментами, но они также представляют собой строительные блоки для создания других инструментов. Проиллюстрируем это на примере. Предположим, требуется написать программу, которая дает возможность пользователю вводить слова. В конце работы программы желательно записать эти слова, как они были введены, получить список использованных слов алфавитном порядке (без учета регистра букв) и вывести количество случаев ввода каждого слова. Для упрощения предположим, что пользовательский ввод не будет содержать цифр и знаков препинания.

Ввод и сохранение списка слов достаточно прост. Руководствуясь примерами из листингов 16.8 и 16.9, можно создать объект `vector<string>` и применить `push_back()` для добавления введенных слов в вектор:

```
vector<string> words;
string input;
while (cin >> input && input != "quit")
   words.push_back(input);
```

А как насчет получения алфавитного списка слов? Можно воспользоваться `sort()`, а затем `unique()`, но такой подход ведет к перезаписи исходных данных, поскольку `sort()` — алгоритм, работающий "по месту". Существует более простой способ, который позволяет избежать этой проблемы. Можно создать объект `set<string>` и скопировать (используя итератор вставки) слова из вектора в набор. Набор автоматически сортирует свое содержимое, что исключает необходимость вызова `sort()`. Кроме того, набор хранит только по одной копии каждого ключа, поэтому не придется вызывать и `unique()`. Но минутку! Спецификация требует игнорировать регистр. Один из способов обеспечения этого — использование `transform()` вместо `copy()` для копирования данных из вектора в набор. В качестве функции трансформации можно применить такую, которая преобразует строки в нижний регистр:

```
set<string> wordset;
transform(words.begin(), words.end(),
   insert_iterator<set<string> > (wordset, wordset.begin()), ToLower);
```

Написание функции `ToLower()` не представляет особой сложности. Нужно просто с помощью `transform()` применить функцию `tolower()` к каждому элементу строки, указывая строку и в качестве источника, и в качестве места назначения. Помните, что объекты `string` также могут использовать функции STL. Передача и возврат строки в качестве ссылки означает, что алгоритм работает с исходной строкой без необходимости создания копии. Вот код функции `ToLower()`:

```
string & ToLower(string & st)
{
   transform(st.begin(), st.end(), st.begin(), tolower);
   return st;
}
```

Одна из потенциальных проблем состоит в том, что функция `tolower()` определяется как `int tolower(int)`, а некоторые компиляторы требуют, чтобы функция соответствовала типу элемента, которым является `char`. Одно из возможных решений — замена `tolower` на `toLower` и предоставление следующего определения:

```
char toLower(char ch) { return tolower(ch); }
```

Для получения количества появлений каждого слова в строке ввода можно применить функцию count(). Она принимает в качестве аргументов диапазон и значение, а возвращает количество появлений этого значения в диапазоне. Можно воспользоваться объектом vector, чтобы задать диапазон, и объектом set, чтобы передать список слов для подсчета. То есть для каждого слова в наборе можно подсчитать количество его появлений в векторе. Чтобы результаты подсчета оставались связанными с соответствующим словом, слово и количество можно сохранить в виде объекта pair<const string, int> внутри объекта map. Слово будет ключом (только одной копией), а количество — значением. Это можно сделать в единственном цикле:

```
map<string, int> wordmap;
set<string>::iterator si;
for (si = wordset.begin(); si != wordset.end(); si++)
   wordmap.insert(pair<string, int>(*si, count(words.begin(),
   words.end(), *si)));
```

Класс map обладает интересной особенностью: для получения доступа к хранимым значениям можно использовать нотацию массивов с ключами в качестве индексов. Например, wordmap["the"] представляло бы значение, связанное с ключом "the", что в данном случае означает количество появлений строки "the" во введенном тексте. Поскольку контейнер wordset содержит все ключи, используемые wordmap, следующий код может выступать альтернативным и более привлекательным способом для сохранения результатов:

```
for (si = wordset.begin(); si != wordset.end(); si++)
   wordmap[*si] = count(words.begin(), words.end(), *si);
```

Так как si указывает на строку в контейнере wordset, *si — это строка, которая может служить ключом для wordmap. Этот код помещает ключи и значения в карту wordmap. Аналогично, нотацию массивов можно применить для выдачи результатов:

```
for (si = wordset.begin(); si != wordset.end(); si++)
   cout << *si << ": " << wordmap[*si] << endl;
```

Если ключ является недопустимым, то соответствующее ему значение будет равно 0.

Программа, представленная в листинге 16.19, объединяет эти идеи и включает в себя код для отображения содержимого этих трех контейнеров (вектора с вводом, набора со списком слов и карты с числом слов).

Листинг 16.19. usealgo.cpp

```
// usealgo.cpp -- использование нескольких элементов STL
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <map>
#include <iterator>
#include <algorithm>
#include <cctype>
using namespace std;

char toLower(char ch) { return tolower(ch); }
string & ToLower(string & st);
void display(const string & s);
int main()
{
   vector<string> words;
   cout << "Enter words (enter quit to quit):\n";  // запрос на ввод слов
```

```
    string input;
    while (cin >> input && input != "quit")
        words.push_back(input);

    cout << "You entered the following words:\n";   // отображение введенных слов
    for_each(words.begin(), words.end(), display);
    cout << endl;

    // Помещение слов в набор с преобразование букв в строчные
    set<string> wordset;
    transform(words.begin(), words.end(),
        insert_iterator<set<string> > (wordset, wordset.begin()),
        ToLower);
    cout << "\nAlphabetic list of words:\n";       // список слов в алфавитном порядке
    for_each(wordset.begin(), wordset.end(), display);
    cout << endl;

    // Помещение и частоты его помещения в карту
    map<string, int> wordmap;
    set<string>::iterator si;
    for (si = wordset.begin(); si != wordset.end(); si++)
        wordmap[*si] = count(words.begin(), words.end(), *si);

    // Отображение содержимого карты
    cout << "\nWord frequency:\n";                 // частота появления слов
    for (si = wordset.begin(); si != wordset.end(); si++)
        cout << *si << ": " << wordmap[*si] << endl;

    return 0;
}

string & ToLower(string & st)
{
    transform(st.begin(), st.end(), st.begin(), toLower);
    return st;
}

void display(const string & s)
{
    cout << s << " ";
}
```

Ниже приведен пример запуска программы из листинга 16.19:

```
Enter words (enter quit to quit):
The dog saw the cat and thought the cat fat
The cat thought the cat perfect
quit
You entered the following words:
The dog saw the cat and thought the cat fat The cat thought the cat perfect

Alphabetic list of words:
and cat dog fat perfect saw the thought

Word frequency:
and: 1
cat: 4
dog: 1
fat: 1
perfect: 1
saw: 1
the: 5
thought: 2
```

Мораль этой демонстрации в том, что при использовании STL следует максимально избегать написания собственного кода. Обобщенное и гибкое проектное решение, положенное в основу STL, должно сэкономить массу работы. Кроме того, разработчики STL – люди, мыслящие алгоритмически и стремящиеся к высокой эффективности. Таким образом, алгоритмы хорошо подобраны и отлажены.

Другие библиотеки

Язык C++ предлагает ряд других библиотек классов, которые в большей степени специализированы, нежели примеры, приведенные в настоящей главе. Например, заголовочный файл `complex` предоставляет шаблонный класс `complex` для комплексных чисел, имеющий отдельные варианты для типов `float`, `long` и `long double`. Этот класс включает стандартные операции с комплексными числами, а также стандартные функции, которые могут быть применены к комплексным числам. Заголовочный файл `random` в C++11 расширяет функциональность работы со случайными числами.

В главе 14 рассматривался класс шаблона `vararray`, поддерживаемый заголовочным файлом `vararray`. Этот класс шаблона спроектирован для представления числовых массивов и предоставляет поддержку множества операций с числовыми массивами, такие как добавление содержимого одного массива к другому, применение математических функций к каждому элементу массива и применение к массивам операций линейной алгебры.

`vector`, `valarray` и `array`

Возможно, вас удивит, почему в C++ определены три шаблона массивов: `vector`, `valarray` и `array`. Эти классы создавались разными группами разработчиков для различных целей. Класс шаблона `vector` является частью системы контейнерных классов и алгоритмов. Класс `vector` поддерживает контейнерно-ориентированные действия вроде сортировки, вставки, переупорядочивания, поиска и передачи данных в другие контейнеры и другие манипуляции с данными. С другой стороны, класс шаблона `valarray` ориентирован на вычислительные операции, и не является частью STL. Например, он не имеет методов `push_back()` и `insert()`, но предоставляет простой интуитивно понятный интерфейс для многих математических операций. И, наконец, `array` разработан в качестве замены встроенного типа массива, сочетая в себе компактность и эффективность типа с более удобным и безопасным интерфейсом. Имея фиксированный размер, `array` не поддерживает `push_back()` и `insert()`, но предоставляет ряд других методов STL. В их число входят `begin()`, `end()`, `rbegin()` и `rend()`, что упрощает применение алгоритмов STL к объектам `array`.

Например, пусть имеются следующие объявления:

```
vector<double> ved1(10), ved2(10), ved3(10);
array<double, 10> vod1, vod2, vod3;
valarray<double> vad1(10), vad2(10), vad3(10);
```

Предположим, что `ved1`, `ved2`, `vod1`, `vod2`, `vad1` и `vad2` получили соответствующие значения. Требуется присвоить сумму первых элементов двух массивов первому элементу третьего массива, и т.д. Используя класс `vector`, необходимо было бы выполнить следующее:

```
transform(ved1.begin(), ved1.end(), ved2.begin(), ved3.begin(),
        plus<double>());
```

Это же можно сделать с классом `array`:

```
transform(vod1.begin(), vod1.end(), vod2.begin(), vod3.begin(),
        plus<double>());
```

Однако класс valarray перегружает все арифметические операции для работы с объектами valarray, поэтому нужно было использовать следующий оператор:

```
vad3 = vad1 + vad2;        // операция + перегружена
```

Аналогично, следующий оператор приводит к тому, что каждый элемент vad3 является произведением соответствующих элементов из vad1 и vad2:

```
vad3 = vad1 * vad2;        // операция * перегружена
```

Допустим, что требуется заменить каждое значение массива им же, но умноженным на 2.5. В STL применяется следующий подход:

```
transform(ved3.begin(), ved3.end(), ved3.begin(),
          bind1st(multiplies<double>(), 2.5));
```

Класс valarray перегружает операцию умножения объекта valarray на одиночное значение, а также различные операции присваивания с вычислением, поэтому можно было бы воспользоваться любым из приведенных ниже операторов:

```
vad3 = 2.5 * vad3;         // операция * перегружена
vad3 *= 2.5;               // операция *= перегружена
```

Предположим, что требуется вычислить натуральный логарифм каждого элемента в массиве и сохранить результат в соответствующем элементе второго массива. В STL применяется следующий подход:

```
transform(ved1.begin(), ved1.end(), ved3.begin(), log);
```

Класс valarray перегружает обычные математически функции для принятия объекта valarray в качестве аргумента и возврата объекта valarray, поэтому можно использовать такой оператор:

```
vad3 = log(vad1);            // операция log() перегружена
```

Или же можно воспользоваться методом apply(), который работает также для неперегруженных функций:

```
vad3 = vad1.apply(log);
```

Метод apply() не изменяет вызывающий объект; вместо этого он возвращает новый объект, содержащий результирующие значения.

Простота интерфейса valarray становится еще более очевидной при выполнении многошаговых вычислений:

```
vad3 = 10.0* ((vad1 + vad2) / 2.0 + vad1 * cos(vad2));
```

Решение этой же задачи с помощью версии vector из STL оставляется в качестве упражнения для самостоятельной проработки.

Класс valarray также предлагает метод sum(), который суммирует содержимое объекта valarray, метод size(), подсчитывающий количество элементов, метод max(), который возвращает наибольшее значение объекта, и метод min(), возвращающий наименьшее значение.

Как видите, valarray обладает явным преимуществом перед vector с точки зрения нотации, если речь идет о математических операциях, однако он менее универсален. Класс valarray имеет метод resize(), но не осуществляет автоматического изменения размера, подобного тому, которое обеспечивает метод push_back() класса vector. Не существует никаких методов для вставки значений, выполнения поиска, сортировки и тому подобных действий. Короче говоря, класс valarray более ограничен, чем класс vector, но его более узкое назначение позволяет иметь намного более простой интерфейс.

Приводит ли более простой интерфейс, предоставляемый valarray, к более высокой производительности? В большинстве случаев нет. Как правило, простая нотация реализуется какими-либо циклами, которые использовались бы с обычными массивами. Однако некоторые разработчики оборудования обеспечивают параллельное выполнение векторных операций, в которых значения массива загружаются в массив регистров. В принципе, операции valarray можно было бы реализовать так, чтобы они пользовались преимуществами такого проектного решения.

Можно ли применять STL для работы с объектами valarray? Для ответа на этот вопрос нужно еще раз вспомнить некоторые принципы STL. Предположим, что имеется объект valarray<double>, содержащий 10 элементов:

```
valarray<double> vad(10);
```

После того как массив заполнен числовыми значениями, можно ли применить к нему функцию сортировки STL? Класс valarray не имеет методов begin() и end(), так что их нельзя использовать в качестве аргументов, задающих диапазон:

```
sort(vad.begin(), vad.end());      // НЕЛЬЗЯ, нет ни begin(), ни end()
```

Кроме того, vad — это объект, а не указатель, поэтому нельзя имитировать использование обычного массива и применять vad и vad + 10:

```
sort(vad, vad + 10);               // НЕТ, vad — это объект, а не адрес
```

Можно использовать операцию получения адреса:

```
sort(&vad[0], &vad[10]);           // может быть?
```

Но поведение операции обращения к индексу, значение которого на единицу превышает значение последнего индекса массива, для valarray не определено. Это не означает, что &vad[10] обязательно не сработает. (На самом деле это работает во всех шести компиляторах, использованных для тестирования этого кода.) Но это значит, что она может не работать. Чтобы этот код привел к сбою, возможно, понадобится весьма маловероятное сочетание условий, такое как пересечение массива с концом блока памяти, зарезервированного для кучи. Но если от вашего кода зависит дорогостоящий проект, то не стоит рисковать получением подобного сбоя.

C++11 исправляет ситуацию, предоставляя шаблонные функции begin() и end(), которые принимают объект valarray в качестве аргумента. Поэтому вместо vad.begin() следует применять begin(vad). Эти функции возвращают значения, которые совместимы с требованиями диапазона STL:

```
sort(begin(vad), end(vad));        // исправление C++11
```

Код в листинге 16.20 иллюстрирует некоторые из относительных преимуществ классов vector и valarray. В нем используется метод push_back() и средства автоматического изменения размера vector для накопления данных. После сортировки чисел программа копирует их из объекта vector в объект valarray того же размера и выполняет несколько математических операций.

Листинг 16.20. valvect.cpp

```
// valvect.cpp -- сравнение vector и valarray
#include <iostream>
#include <valarray>
#include <vector>
#include <algorithm>
int main()
{
```

```
    using namespace std;
    vector<double> data;
    double temp;
    cout << "Enter numbers (<=0 to quit):\n";  // запрос на ввод положительных чисел
    while (cin >> temp && temp > 0)
        data.push_back(temp);
    sort(data.begin(), data.end());
    int size = data.size();
    valarray<double> numbers(size);
    int i;
    for (i = 0; i < size; i++)
        numbers[i] = data[i];
    valarray<double> sq_rts(size);
    sq_rts = sqrt(numbers);
    valarray<double> results(size);
    results = numbers + 2.0 * sq_rts;
    cout.setf(ios_base::fixed);
    cout.precision(4);
    for (i = 0; i < size; i++)
    {
        cout.width(8);
        cout << numbers[i] << ": ";
        cout.width(8);
        cout << results[i] << endl;
    }
    cout << "done\n";
    return 0;
}
```

Ниже приведен пример запуска программы из листинга 16.20:

```
Enter numbers (<=0 to quit):
3.3 1.8 5.2 10 14.4 21.6 26.9 0
   1.8000:    4.4833
   3.3000:    6.9332
   5.2000:    9.7607
  10.0000:   16.3246
  14.4000:   21.9895
  21.6000:   30.8952
  26.9000:   37.2730
done
```

Класс `valarray` имеет множество возможностей помимо тех, что уже были описаны. Например, если `numbers` — объект `valarray<double>`, то следующий оператор создает массив значений типа `bool`, в котором `vbool[i]` устанавливается равным значению `numbers[i] > 9`, т.е. `true` или `false`:

```
valarray<bool> vbool = numbers > 9;
```

Существуют расширенные версии индексации. Рассмотрим одну из них — класс `slice`, представляющий срез. Объект класса `slice` может использоваться в качестве индекса массива — в этом случае он представляет не просто одно значение, а некоторый поднабор значений. Объект `slice` инициализируется тремя целочисленными значениями: *началом*, *количеством* и *шагом*. *Начало* указывает индекс первого элемента, который должен быть выбран, *количество* задает число выбираемых элементов, а *шаг* представляет расстояние между соседними элементами. Например, объект, сконструированный как `slice(1,4,3)`, означает выбор четырех элементов с индексами 1, 4, 7 и 10.

Другими словами, нужно начать со стартового элемента, добавить шаг для получения следующего элемента и т.д. до тех пор, пока не будет выбрано 4 элемента. Если, скажем, variant — это объект valarray<int>, то следующий оператор присвоит элементам 1, 4, 7 и 10 значение 10:

```
varint[slice(1,4,3)] = 10;  // присваивание выбранным элементам значения 10
```

Это специальное свойство индексации позволяет применять одномерный объект valarray для представления двумерных данных. Например, предположим, что требуется представить массив из 4 строк и 3 столбцов. Информацию можно сохранить в 12-элементном объекте valarray. Тогда объект slice(0,3,1), использованный в качестве индекса, представлял бы элементы 0, 1 и 2 — т.е. первую строку. Аналогично индекс slice(0,4,3) представлял бы элементы 0, 3, 6 и 9 — т.е. первый столбец. Некоторые возможности slice демонстрируются в листинге 16.21.

Листинг 16.21. vslice.cpp

```
// vslice.cpp -- использование срезов valarray
#include <iostream>
#include <valarray>
#include <cstdlib>

const int SIZE = 12;
typedef std::valarray<int> vint;         // для упрощения объявлений
void show(const vint & v, int cols);
int main()
{
   using std::slice;                     // из <valarray>
   using std::cout;
   vint valint(SIZE);                    // представляет 4 строки по 3 элемента
   int i;
   for (i = 0; i < SIZE; ++i)
      valint[i] = std::rand() % 10;
   cout << "Original array:\n";          // исходный массив
   show(valint, 3);                      // отображение в виде 3 столбцов
   vint vcol(valint[slice(1,4,3)]);      // извлечение 2-го столбца
   cout << "Second column:\n";
   show(vcol, 1);                        // отображение в 1 столбце
   vint vrow(valint[slice(3,3,1)]);      // извлечение 2-ой строки
   cout << "Second row:\n";
   show(vrow, 3);
   valint[slice(2,4,3)] = 10;            // присваивание 2-му столбцу
   cout << "Set last column to 10:\n";
   show(valint, 3);
   cout << "Set first column to sum of next two:\n";

   // Операция + не определена для slice, поэтому преобразуем в valarray<int>
   valint[slice(0,4,3)] = vint(valint[slice(1,4,3)])
                        + vint(valint[slice(2,4,3)]);
   show(valint, 3);
   return 0;
}
void show(const vint & v, int cols)
{
   using std::cout;
   using std::endl;
   int lim = v.size();
   for (int i = 0; i < lim; ++i)
   {
```

```
        cout.width(3);
        cout << v[i];
        if (i % cols == cols - 1)
            cout << endl;
        else
        cout << ' ';
    }
    if (lim % cols != 0)
        cout << endl;
}
```

Операция + определена для объектов valarray, таких как valint, и определена для отдельного элемента int вроде valint[1], но, как отмечено в коде листинга 16.19, операция + не определена для проиндексированных с помощью slice единиц valarray наподобие valint[slice(1,4,3)]. Поэтому, чтобы сделать возможным сложение, программа конструирует из срезов полные объекты:

```
vint(valint[slice(1,4,3)])    // вызов конструктора на основе slice
```

Класс valarray предлагает конструкторы для этой цели.

Ниже показан пример запуска программы из листинга 16.21:

```
Original array:
  0 3 3
  2 9 0
  8 2 6
  6 9 1
Second column:
  3
  9
  2
  9
Second row:
  2 9 0
Set last column to 10:
  0 3 10
  2 9 10
  8 2 10
  6 9 10
Set first column to sum of next two:
 13 3 10
 19 9 10
 12 2 10
 19 9 10
```

Поскольку значения устанавливаются с использованием rand(), разные реализации rand() приведут к разным значениям.

Существуют и другие возможности, включая класс gslice, предназначенный для представления многомерных массивов, но сказанного должно быть достаточно, чтобы дать представление о том, что собой представляет valarray.

Шаблон initializer_list (C++11)

Шаблон initializer_list является еще одним дополнением C++11 к библиотеке C++. Синтаксис списка инициализаторов можно использовать для инициализации контейнера STL списком значений:

```
std::vector<double> payments {45.99, 39.23, 19.95, 89.01};
```

Этот оператор создает контейнер для четырех элементов, инициализируя их четырьмя значениями из списка. Подобное возможно благодаря тому, что теперь классы контейнеров имеют конструкторы, которые принимают аргумент initializer_list<T>. Например, объект vector<double> имеет конструктор, который принимает аргумент initializer_list<double>, и предыдущее объявление эквивалентно следующему:

```
std::vector<double> payments({45.99, 39.23, 19.95, 89.01});
```

В этом примере список явно записан в виде аргумента конструктора.

Обычно с помощью универсального синтаксиса инициализации C++11 класс конструктора можно вызвать, используя нотацию {} вместо ():

```
shared_ptr<double> pd {new double};  // можно использовать {} вместо ()
```

Но это приводит к проблеме при наличии конструктора initializer_list:

```
std::vector<int> vi{10};       // ??
```

Какой конструктор вызывает этот оператор?

```
std::vector<int> vi(10);       // случай A: 10 неинициализированных элементов
std::vector<int> vi({10});     // случай B: первый элемент установлен в 10
```

Ответ следующий: если класс обладает конструктором, который принимает аргумент initializer_list, то применение синтаксиса {} ведет к вызову конкретного конструктора. Поэтому к данному примеру применим случай B.

Все элементы initializer_list должны быть одного типа. Однако компилятор будет выполнять преобразования для приведения типа в соответствие:

```
std::vector<double> payments {45.99, 39.23, 19, 89};
// то же что и std::vector<double> payments {45.99, 39.23, 19.0, 89.0};
```

Поскольку в этом примере типом элементов vector является double, типом списка будет initializer_list<double>, и значения 19 и 89 преобразуются в тип double. При этом применяются обычные ограничения списка, налагаемые на его сужение:

```
std::vector<int> values = {10, 8, 5.5}; // сужение, ошибка времени компиляции
```

В данном случае тип элемента — int, а неявное преобразование 5.5 к типу int не допускается.

Нет смысла указывать конструктор initializer_list, если только класс не предназначен для работы со списками переменных размеров. Например, нежелательно использовать конструктор initializer_list для класса, который принимает фиксированное число значений. Приведенное ниже объявление не предоставляет конструктор initializer_list для трех членов данных:

```
class Position
{
private:
   int x;
   i    n    t         y    ;
   i    n    t         z    ;
public:
   Position(int xx = 0, int yy = 0, int zz = 0)
          : x(xx), y(yy), z(zz) {}
   // нет конструкторов initializer_list
   ...
};
```

Это позволяет применять синтаксис {} с конструктором Position(int,int,int):

```
Position A = {20, -3};  // использует Position(20,-3,0)
```

Использование initializer_list

Объекты initializer_list можно применять в коде, подключая заголовочный файл initializer_list. Класс шаблона имеет члены begin() и end(), и их можно использовать для получения доступа к элементам списка. Класс имеет также член size(), который возвращает количество элементов. Простой пример использования initializer_list приведен в листинге 16.22. Он требует применения компилятора, который поддерживает это средство C++11.

Листинг 16.22. list.cpp

```
// ilist.cpp -- использование initializer_list (средство C++11)
#include <iostream>
#include <initializer_list>

double sum(std::initializer_list<double> il);
double average(const std::initializer list<double> & ril);

int main()
{
   using std::cout;
   cout << "List 1: sum = " << sum({2,3,4})
        <<", ave = " << average({2,3,4}) << '\n';  // список 1, его сумма и среднее

   std::initializer_list<double> dl = {1.1, 2.2, 3.3, 4.4, 5.5};
   cout << "List 2: sum = " << sum(dl)
        <<", ave = " << average(dl) << '\n';       // список 2, его сумма и среднее

   dl = {16.0, 25.0, 36.0, 40.0, 64.0};
   cout << "List 3: sum = " << sum(dl)
        <<", ave = " << average(dl) << '\n';       // список 3, его сумма и среднее
   return 0;
}

double sum(std::initializer_list<double> il)
{
   double tot = 0;
   for (auto p = il.begin(); p !=il.end(); p++)
      tot += *p;
   return tot;
}

double average(const std::initializer_list<double> & ril)
{
   double tot = 0;
   int n = ril.size();
   double ave = 0.0;

   if (n > 0)
   {
      for (auto p = ril.begin(); p !=ril.end(); p++)
         tot += *p;
      ave = tot / n;
   }

   return ave;
}
```

Ниже приведен пример запуска программы из листинга 16.22:

```
List 1: sum = 9, ave = 3
List 2: sum = 16.5, ave = 3.3
List 3: sum = 181, ave = 36.2
```

Замечания по программе

Объект `initializer_list` можно передавать по значению или по ссылке, как продемонстрировано в функциях `sum()` и `average()`. Сам по себе объект мал и, как правило, состоит из двух указателей (одного на начало, и второго — на элемент, следующий за последним) или указателя на начало и целочисленного значения, представляющего размер, поэтому выбор конкретного способа не имеет особого значения для производительности. (В STL они передаются по значению.)

Аргументом функции может быть литерал списка, подобный `{2,3,4}` или переменная списка вроде `dl`.

Типом итераторов для `initializer_list` является `const`, поэтому изменение значений в списке невозможно:

```
*dl.begin() = 2011.6;                    // не разрешено
```

Но, как показано в листинге 16.22, переменную списка можно присоединить к другому списку:

```
dl = {16.0, 25.0, 36.0, 40.0, 64.0};     // разрешено
```

Однако запланированное применение класса `initializer_list` — передача списка значений конструктору или другой функции.

Резюме

Язык C++ включает в себя мощный набор библиотек, предлагающих решения для многих часто встречающихся задач, а также инструменты, упрощающие решение множества других задач. Класс `string` предоставляет удобные средства обработки строк как объектов, а также автоматическое управление памятью и ряд методов и функций для работы со строками. Например, эти методы и функции позволяют осуществлять конкатенацию строк, вставлять одни строки в другие, изменять порядок следования элементов в строках, искать в них символы или подстроки, а также выполнять операции ввода и вывода.

Шаблоны интеллектуальных указателей, такие как `auto_ptr` и `unique_ptr` и `shared_ptr` из C++11, упрощают управление памятью, выделенной операцией `new`. Если использовать один из этих интеллектуальных указателей вместо обычного указателя для хранения адреса, возвращенного `new`, не нужно помнить о необходимости вызова операции `delete`. Когда объект интеллектуального указателя уничтожается, его деструктор автоматически вызывает операцию `delete`.

Библиотека STL — это коллекция шаблонных классов контейнеров, шаблонных классов итераторов, шаблонных функциональных объектов и шаблонных функций алгоритмов, которые построены на основе унифицированного проектного решения, соответствующего принципам обобщенного программирования. Алгоритмы используют шаблоны для обеспечения их общности в плане хранимых типов и интерфейса итераторов, что обеспечивает их общность в плане типа контейнеров. Итераторы — это обобщения указателей.

Для обозначения набора требований в STL применяется термин *концепция*. Например, концепция однонаправленных итераторов включает требования, в соот-

ветствии с которыми объект однонаправленного итератора может быть разыменован для чтения и записи, а также инкрементирован. Действительную реализацию концепции называют *моделированием* концепции. Например, концепция однонаправленного итератора может быть смоделирована обыкновенным указателем либо объектом, предназначенным для навигации по связному списку. Концепции, основанные на других концепциях, называются *уточнениями*. Например, двунаправленный итератор — это уточнение концепции однонаправленного итератора.

Классы контейнеров, подобные `vector` и `set`, являются моделями концепций контейнеров, таких как контейнеры, последовательности и ассоциативные контейнеры. В библиотеке STL определено несколько шаблонов классов контейнеров: `vector`, `deque`, `list`, `set`, `multiset`, `map`, `multimap` и `bitset`. В ней также определены шаблоны классов адаптеров: `queue`, `priority_queue` и `stack`; эти классы адаптируют лежащие в их основе классы контейнеров, предоставляя им характерный интерфейс, предполагаемый именем соответствующего шаблона класса адаптера. Так, хотя по умолчанию класс `stack` и основан на классе `vector`, он допускает вставку и извлечение элементов только с вершины стека. C++11 добавляет классы `forward_list`, `unordered_set`, `unordered_multiset`, `unordered_map` и `unordered_multimap`.

Некоторые алгоритмы выражены в виде методов контейнерных классов, но большинство — в форме общих автономных (не являющихся членами класса) функций. Это обеспечивается использованием итераторов в качестве интерфейса между контейнерами и алгоритмами. Одно из преимуществ такого подхода состоит в том, что нужна только одна функция `for_each()` или `copy()` вместо отдельных версий для каждого из контейнеров. Второе преимущество заключается в том, что алгоритмы STL могут применяться с контейнерами, не входящими в состав STL, такими как обычные массивы, объекты `string`, объекты `array` или же любые классы, разработанные как совместимые с итераторами STL и идиомой контейнера.

И контейнеры, и алгоритмы характеризуются типом итераторов, которые они представляют либо в которых нуждаются. Необходимо удостовериться, что контейнер поддерживает концепцию итератора, служащую нуждам соответствующих алгоритмов. Например, алгоритм `for_each()` использует входной итератор, чьи минимальные требования удовлетворяются всеми типами классов контейнеров STL. Однако `sort()` требует итераторов произвольного доступа, который поддерживают не все классы контейнеров. Класс контейнера может предоставить специализированный метод в качестве варианта для выбора, если он не отвечает требованиям определенного алгоритма. Например, класс `list` обладает методом `sort()`, который основан на двунаправленных итераторах, и поэтому может применять этот метод вместо общей функции.

Библиотека STL предлагает также функциональные объекты, или функторы, которые представляют собой классы с перегруженной операцией `()`, т.е. те, для которых определен метод `operator()()`.

Объекты таких классов могут быть вызваны с использованием функциональной нотации, но могут нести в себе дополнительную информацию. Например, адаптируемые функторы содержат операторы `typedef`, которые идентифицируют типы аргументов и тип возвращаемого значения функтора. Эта информация может быть использована другими компонентами, такими как адаптеры функций.

Представляя общие контейнерные типы и множество общих операций, реализованных с помощью эффективных алгоритмов, причем все это выполняется все в обобщенной манере, библиотека STL превращается в прекрасный источник повторно используемого кода. Программные задачи можно решать непосредственно с помощью

инструментов STL либо применять их в качестве строительных блоков для построения нужных решений.

Классы шаблонов `complex` и `valarray` поддерживают численные операции с массивами и комплексными числами.

Вопросы для самоконтроля

1. Пусть имеется следующее объявление класса:

```
class RQ1
{
private:
    char * st;      // указатель на строку в стиле C
public:
    RQ1() { st = new char [1]; strcpy(st,""); }
    RQ1(const char * s)
    {st = new char [strlen(s) + 1]; strcpy(st, s); }
    RQ1(const RQ1 & rq)
    {st = new char [strlen(rq.st) + 1]; strcpy(st, rq.st); }
    ~RQ1() {delete [] st};
    RQ & operator=(const RQ & rq);
    // Другие операторы
};
```

   Преобразуйте его в объявление, использующее объект `string`. Какие методы больше не нуждаются в явных определениях?

2. Назовите хотя бы два преимущества объектов `string` по сравнению со строками в стиле C с точки зрения простоты применения.

3. Напишите функцию, которая принимает ссылку на объект `string` в качестве аргумента и затем преобразует объект `string` в прописные буквы.

4. Какие из следующих операторов не являются примерами корректного использования (концептуально или синтаксически) объекта `auto_ptr`? (Предполагается, что все необходимые заголовочные файлы включены.)

```
auto_ptr<int> pia(new int[20]);
auto_ptr<string> (new string);
int rigue = 7;
auto_ptr<int>pr(&rigue);
auto_ptr dbl (new double);
```

5. Если бы можно было создать механический эквивалент стека, который хранит клюшки для гольфа вместо номеров, почему он был бы (концептуально) плохой сумкой для гольфа?

6. Почему контейнер `set` — неудачный выбор для хранения записей о полученных очках в гольфе в формате "от лунки к лунке"?

7. Если указатель — это итератор, почему разработчики STL просто не используют его вместо итераторов?

8. Почему разработчики STL не определили базовый класс итератора, используя наследование для порождения классов для других типов итераторов, и не выразили алгоритмы в терминах этих классов итераторов?

9. Приведите, как минимум, три примера, показывающих преимущества объекта `vector` по сравнению с обычным массивом.

10. Если в коде из листинга 16.9 использовать `list` вместо `vector`, то какие части программы станут некорректными? Легко ли они могут быть исправлены? Если да, то как?

Упражнения по программированию

1. *Палиндром* — это строка, которая читается одинаково в обоих направлениях. Например, "tot" и "otto" — короткие палиндромы. Напишите программу, которая запрашивает у пользователя строку и передает ссылку на нее функции `bool`. Функция должна возвращать `true`, если строка является палиндромом, и `false` — в противном случае. Пока не беспокойтесь о таких нюансах, как прописные и строчные буквы, пробелы и знаки препинания. Другими словами, эта простая версия должна отклонять строки типа "Otto" и "Madam, I'm Adam". Для упрощения решения задачи можете обратиться к списку строковых методов, приведенному в приложении E.

2. Решите задачу из упражнения 1, но учтите такие нюансы, как прописные и строчные буквы, пробелы и знаки препинания. То есть строка "Madam, I'm Adam" должна быть признана палиндромом. Например, функция проверки могла бы уменьшить строку до "madamimadam", а затем проверить, эквивалентна ли ей строка в обратном порядке. Не забудьте об удобной библиотеке `cctype`. В ней можно найти несколько подходящих функций STL, хотя это и не обязательно.

3. Измените программу из листинга 16.3, чтобы она получала слова из файла. Один из возможных подходов предполагает использование объекта `vector<string>` вместо массива элементов типа `string`. Затем можно применить `push_back()` для копирования стольких слов, сколько их имеется в файле данных, в объект `vector<string>`, а с помощью функции-члена `size()` определить длину списка слов. Поскольку программа должна считывать слова из файла по одному, необходимо использовать операцию >>, а не `getline()`. Сам файл должен содержать слова, разделенные пробелами, символами табуляции или символами новой строки.

4. Напишите функцию с интерфейсом в старом стиле, которая имеет следующий прототип:

```
int reduce(long ar[], int n);
```

   Действительными аргументами должны быть имя массива и количеством элементов в нем. Функция должна сортировать массив, удалять дублированные значения и возвращать значение, равное числу элементов в уменьшенном массиве. Напишите эту функцию, используя функции STL. (Если вы решите применить общую функцию `unique()`, обратите внимание, что она возвращает конец результирующего диапазона.) Протестируйте эту функцию в короткой программе.

5. Решите ту же задачу, что и в упражнении 4, но с помощью шаблонной функции:

```
template <class T>
int reduce(T ar[], int n);
```

   Протестируйте функцию в короткой программе, используя экземпляры с `long` и `string`.

6. Повторите пример, показанный в листинге 12.12, используя шаблон `queue` класса STL вместо класса `Queue`, который был описан в главе 12.

7. Лотерея — довольно популярная игра. Карточка лотереи имеет нумерованные поля, из которых случайным образом выбирается определенное количество номеров. Напишите функцию `Lotto()`, принимающую два аргумента. Первым должно быть число номеров на карточке лотереи, а вторым — количество случайным образом выбранных номеров. Функция должна возвращать объект `vector<int>`, содержащий отсортированные по порядку случайно выбранные номера. Эту функцию можно использовать, например, так:

```
vector<int> winners;
winners = Lotto(51,6);
```

Этот код присвоил бы объекту `winners` вектор, содержащий шесть случайным образом выбранных номеров в диапазоне от 1 до 51. Обратите внимание, что простого использования `rand()` не достаточно для решения этого упражнения, потому что она может генерировать дублированные значения. Совет: пусть функция создает вектор, который содержит все возможные значения, затем применяйте `random_shuffle()`, после чего используйте начало перетасованного вектора для получения значений. Также напишите короткую программу для тестирования разработанной функции.

8. Мэт и Пэт хотят пригласить своих друзей на вечеринку. Они просят вас написать программу, которая делает следующее.
   - Позволяет Мэту ввести список имен его друзей. Имена сохраняются в контейнере и затем отображаются в отсортированном порядке.
   - Позволяет Пэт ввести список ее друзей. Имена сохраняются во втором контейнере и затем отображаются в отсортированном порядке.
   - Создает третий контейнер, который объединяет эти два списка, исключает дубликаты и отображает содержимое этого контейнера.
9. По сравнению с массивом связный список отличается более простым добавлением и удалением элементов, но медленной сортировкой. Поэтому возникает вопрос: возможно, было бы быстрее скопировать список в массив, отсортировать массив и скопировать отсортированный результат обратно в список, чем просто использовать алгоритм списка для сортировки. (Но это может быть связано с наличием большего объема памяти.) Проверьте гипотезу о более быстром выполнении задачи, применив следующий подход.
   а. Создайте большой объект `vi0` типа `vector<int>`, используя `rand()` для задания начальных значений.
   б. Создайте второй объект `vi` типа `vector<int>` и объект `li` типа `list<int>` того же размера, что и исходный, и инициализируйте их значениями исходного вектора.
   в. Замерьте время, требуемое программе для сортировки `vi` с помощью алгоритма `sort()` из STL, а затем время, необходимое для сортировки `li` посредством метода `list sort()`.
   г. Переустановите `li` неотсортированным содержимым `vi0`. Замерьте время выполнения смешанной операции копирования `li` в `vi`, сортировки `vi` и копирования результата обратно в `li`.

Для измерения времени выполнения этих операций можно использовать `clock()` из библиотеки `ctime`. Как показано в листинге 5.14, для запуска первого таймера можно применять следующий оператор:

```
clock_t start = clock();
```

Для получения прошедшего времени в конце операции используйте следующий оператор:

```
clock_t end = clock();
cout << (double)(end - start)/CLOCKS_PER_SEC;
```

Вне всяких сомнений, этот тест показателен, поскольку результаты будут зависеть от ряда факторов, в том числе объема доступной памяти, применения многопроцессорной обработки и размеров массива или списка. (С увеличением количества сортируемых элементов можно ожидать большего увеличения эффективности массива по сравнению со списком.) Кроме того, при наличии выбора между стандартной сборкой и окончательной сборкой, следует выбирать окончательную сборку. В современных высокоскоростных компьютерах для получения репрезентативных результатов необходимо использовать массив максимально возможного размера. Например, можно иметь 100 000, 1 000 000 и 10 000 000 элементов.

10. Измените код в листинге 16.9 (`vect3.cpp`) следующим образом.

    а. Добавьте член `price` в структуру `Review`.

    б. Для хранения вводимых данных используйте вектор (`vector`) объектов `shared_ptr<Review>` вместо вектора объектов `Review`. Помните, что `shared_ptr` должен быть инициализирован указателем, возвращенным операцией `new`.

    в. Вслед за этапом ввода данных реализуйте цикл, который предоставляет пользователю следующие варианты отображения книг: в исходном порядке, в алфавитном порядке, в порядке возрастания рейтинга, в порядке возрастания цены, в порядке уменьшения цены и возможность завершения программы.

    Один из возможных подходов может быть таким. После получения первоначальных введенных данных создайте еще один вектор указателей `shared_ptr`, инициализированный исходным массивом. Определите функцию `operator<()`, которая сравнивает указанные структуры, и примените ее для сортировки второго вектора так, чтобы `shared_ptr` были упорядочены по названиям книг, сохраненных в указанных объектах. Повторите процесс, чтобы получить вектор объектов `shared_ptr`, отсортированных по `rating` и `price`. Обратите внимание, что `rbegin()` и `rend()` избавляют от необходимости создания векторов с обратным порядком следования элементов.

17

Ввод, вывод и файлы

В ЭТОЙ ГЛАВЕ...

- Взгляд C++ на ввод и вывод
- Семейство классов `iostream`
- Перенаправление
- Методы класса `ostream`
- Форматирование вывода
- Методы класса `istream`
- Состояния потока
- Файловый ввод-вывод
- Использование класса `ifstream` для ввода из файлов
- Использование класса `ofstream` для вывода в файлы
- Использование класса `fstream` для файлового ввода и вывода
- Обработка командной строки
- Бинарные файлы
- Произвольный доступ к файлам
- Внутреннее форматирование

Обсуждение ввода и вывода C++ – непростая задача. С одной стороны, практически каждая программа использует ввод и вывод, и изучение их применения – одна из первых задач, стоящих перед теми, кто осваивает язык программирования. С другой стороны, для реализации этих операций C++ применяет многие из своих наиболее развитых языковых средств, включая классы, производные классы, перегрузку функций, виртуальные функции, шаблоны и множественное наследование. Таким образом, чтобы действительно понять ввод-вывод C++, нужно обладать достаточно глубокими знаниями о языке в целом. Чтобы ввести вас в курс дела, в предшествующих главах книги были обрисованы основные способы использования объектов `cin` класса `istream` и объекта `cout` класса `ostream` для ввода и вывода и (в меньшей степени) работу с объектами `ifstream` и `ofstream` для файлового ввода и вывода.

В этой главе более подробно рассмотрены классы ввода и вывода C++, показано, как они спроектированы, и пояснено управление форматом вывода. (Если вы пропустили несколько глав, чтобы быстрее ознакомиться с развитыми средствами форматирования, то можете прочесть разделы, посвященные форматированию, обращая внимание на технические подробности и игнорируя объяснения.)

Средства C++, предназначенные для файлового ввода и вывода, базируются на тех же основных определениях классов, что и объекты `cin` и `cout`, поэтому в основу исследования файловых операций ввода-вывода в данной главе положен консольный ввод-вывод (с клавиатуры и на экран).

Комитет по стандартизации ANSI/ISO C++ предпринял ряд усилий, чтобы сделать ввод-вывод C++ в большей степени совместимым с существующим вводом-выводом языка C, и это привело к некоторым изменениям по сравнению с традиционными подходами C++.

Обзор ввода и вывода в C++

В большинстве языков программирования операции ввода и вывода встроены в сам язык. Например, если вы просмотрите список ключевых слов таких языков, как BASIC и Pascal, то увидите, что операторы `PRINT`, `writeln` и им подобные являются частью словарей этих языков. Но ни C, ни C++ не имеют операций ввода и вывода, встроенных в сам язык. Если просмотреть ключевые слова этих языков, то там обнаружатся, скажем, `for` и `if`, но ничего такого, что имело бы отношение к вводу и выводу. Изначально язык C оставлял ввод-вывод на усмотрение создателей компиляторов. Одна из причин такого подхода состояла в том, чтобы предоставить разработчикам компиляторов определенную свободу в проектировании функций ввода-вывода для обеспечения наилучшего соответствия требованиям оборудования целевой платформы. На практике большинство реализаций ввода-вывода основано на библиотечных функциях, изначально разработанных для среды Unix. ANSI C формализовал спецификации этого пакета в стандарте под названием "Стандартный пакет ввода-вывода" ("Standard Input/Output package"), утвердив его в качестве неотъемлемой составной части стандартной библиотеки C. Язык C++ также распознает этот пакет, поэтому если вы знакомы с семейством функций C, объявленных в файле `stdio.h`, то можете использовать их в программах C++. (Для поддержки этих функций в более новых реализациях используется заголовочный файл `cstdio`).

Однако для организации ввода-вывода C++ полагается на решения C++ вместо решений, предлагаемых C, и это решение представляет собой набор классов, определенных в заголовочных файлах `iostream` (бывший `iostream.h`) и `fstream` (бывший `fstream.h`). Упомянутая библиотека классов не является частью формального определения языка (`cin` и `istream` – это не ключевые слова); в конце концов, язык програм-

мирования определяет правила выполнения таких задач, как создание классов, но не определяет, что именно нужно создавать, следуя этим правилам. Но так же, как реализации C поставляются со стандартными библиотеками функций, C++ поставляется со стандартными библиотеками классов. Изначально стандартная библиотека классов подчинялась требованиям неформального стандарта, который включал только классы, определенные в заголовочных файлах `iostream` и `fstream`. Комитет по стандартизации ANSI/ISO C++ решил формализовать эту библиотеку в качестве стандартной библиотеки классов и добавить еще несколько стандартных классов, вроде тех, что обсуждались в главе 16. Настоящая глава посвящена стандартному вводу-выводу C++. Но для начала мы ознакомимся с концептуальной платформой ввода-вывода C++.

Потоки и буферы

Программа на C++ воспринимает ввод и вывод как потоки байтов. При вводе программа извлекает байты из входного потока, а при выводе помещает байты в выходной поток. Для текстовых программ каждый байт может представлять символ. В общем случае байты могут образовывать двоичное представление символов и числовых данных. Байты входного потока могут поступать с клавиатуры, но также из устройств хранения, вроде жесткого диска, либо из другой программы. Аналогично, байты выходного потока могут передаваться на дисплей, на принтер, на устройство хранения или же отправляться другой программе. Потоки служат посредниками между программой и источником или местом назначения потока. Такой подход позволяет программам на C++ трактовать ввод с клавиатуры так же, как и ввод из файла; программа на C++ просто просматривает поток байтов, не нуждаясь в информации о том, откуда эти байты поступают. Точно так же, используя потоки, программа на C++ может обрабатывать вывод независимо от того, куда, собственно, направляются байты. Таким образом, управление вводом включает две стадии:

- ассоциирование потока с вводом программы;
- подключение потока к файлу.

Другими словами, входной поток нуждается в двух подключениях — по одному на каждой стороне. Подключение со стороны файла представляет собой источник данных для потока, а подключение со стороны программы загружает выходные данные потока в программу. (Подключение со стороны файла может быть файлом, но так же оно может быть устройством, таким как клавиатура.) Аналогично, управление выводом предусматривает подключение выходного потока к программе и ассоциирование некоторого выходного места назначения с этим потоком. Этот процесс можно сравнить с трубопроводом, по которому вместо воды передаются байты (рис. 17.1).

Обычно ввод и вывод может более эффективно обрабатываться посредством буфера. *Буфер* — это блок памяти, используемый в качестве промежуточного временного хранилища при передаче информации от устройства в программу или из программы устройству. Обычно такие устройства, как приводы дисков, передают информацию блоками размером по 512 байт или более, в то время как программы часто обрабатывают данные по одному байту за раз. Буфер облегчает согласование этих двух различных скоростей передачи информации.

Например, предположим, что программа должна подсчитать количество символов доллара в файле на жестком диске. Программа может прочитать один символ из файла, обработать его, прочитать следующий символ из файла и т.д. Чтение из дискового файла по одному символа требует множество операций обращения к оборудованию и выполняется медленно.

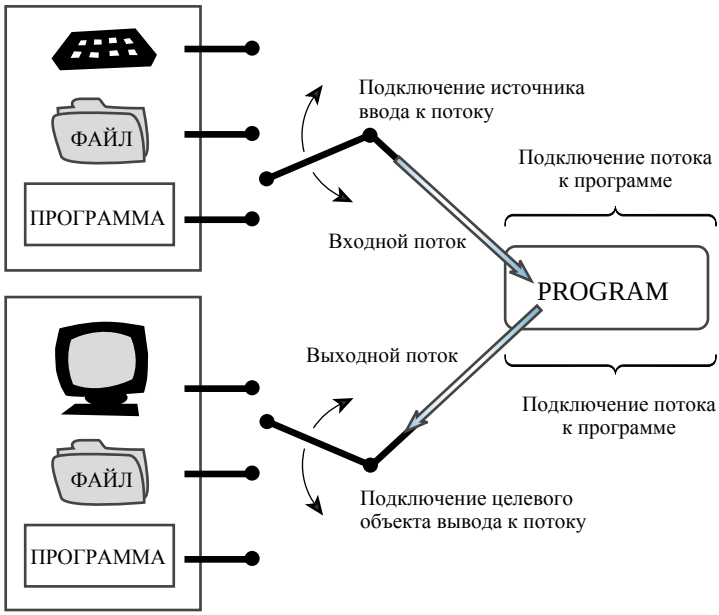
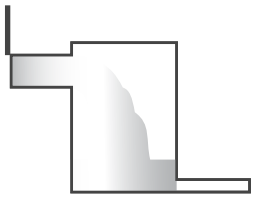


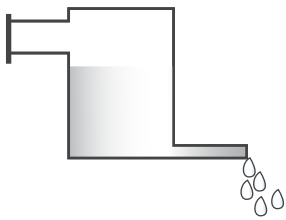
Рис. 17.1. *Ввод и вывод в C++*

Буферизованный подход заключается в чтении большой порции данных с диска, сохранении этой порции в буфере и последующем считывании из буфера по одному символу. Поскольку чтение отдельных байтов из памяти выполняется намного быстрее, чем с жесткого диска, такой подход значительно быстрее и легче для оборудования. Конечно, после того, как программа достигает конца буфера, ей приходится затем считывать следующую порцию данных с диска. Принцип подобен использованию резервуара с водой, накапливающего большой объем дождевой воды во время непогоды, из которого затем вода поступает в дом небольшими порциями по мере необходимости (рис. 17.2). Аналогично, при выводе программа может сначала наполнять буфер, а затем передавать блок данных целиком на жесткий диск, очищая буфер для следующего пакета выходных данных. Этот процесс называют *очисткой буфера*. Можете сами подобрать "водопроводную" аналогию описанному процессу.

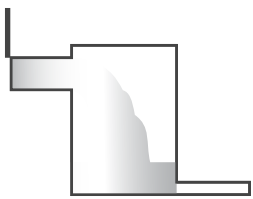
Клавиатурный ввод поставляет символы по одному, поэтому в его случае программа не нуждается в буфере для согласования разных скоростей передачи данных. Однако буферизованный клавиатурный ввод обеспечивает пользователю возможность вернуться назад и исправить ввод до того, как он будет передан в программу. Обычно программа C++ очищает буфер ввода при нажатии клавиши <Enter>. Вот почему примеры в этой книге не начинают обработку данных, пока вы не будет нажата клавиша <Enter>. Для вывода на дисплей программа C++ обычно очищает выходной буфер при передаче символа новой строки. В зависимости от реализации, программа может очищать ввод и в других случаях – например, при предстоящем вводе. То есть, когда программа достигает оператора ввода, она очищает любой вывод, находящийся в выходном буфере. Реализации C++, которые совместимы с ANSI C, должны вести себя таким же образом.



Заполнение буфера потока блоком данных



Выходной поток загружает данные в программу байт за байтом



Повторное заполнение буфера потока следующим блоком данных

***Рис. 17.2.** Поток с буфером*

Потоки, буферы и файл `iostream`

Работа по управлению потоками и буферами может оказаться несколько более сложной, но включение заголовочного файла `iostream` (бывшего `iostream.h`) предоставляет в ваше распоряжение несколько классов, предназначенных для реализации и управления потоками и буферами. Ввод-вывод в C++98 определяет шаблоны классов для поддержки как данных `char`, так и данных `wchar_t`. Ввод-вывод в C++11 добавляет специализации `char16_t` и `char32_t`. Используя средство `typedef`, C++ делает специализацию `char` этих шаблонов подобной традиционным, нешаблонным реализациям ввода-вывода. Ниже представлены некоторые из этих классов (рис. 17.3).

- Класс `streambuf` предоставляет память для буфера, а также и методы для его заполнения, доступа к содержимому, очистки буфера и управления памятью буфера.
- Класс `ios_base` представляет общие свойства потока, такие как признак того, открыт ли поток для чтения, и является он бинарным или текстовым.
- Класс `ios` базируется на `ios_base` и включает член-указатель на объект класса `streambuf`.
- Класс `ostream` является производным от `ios` и предоставляет методы вывода.
- Класс `istream` является производным от `ios` и предоставляет методы ввода.
- Класс `iostream` базируется на классах `istream` и `ostream` и потому наследует как методы ввода, так и методы вывода.

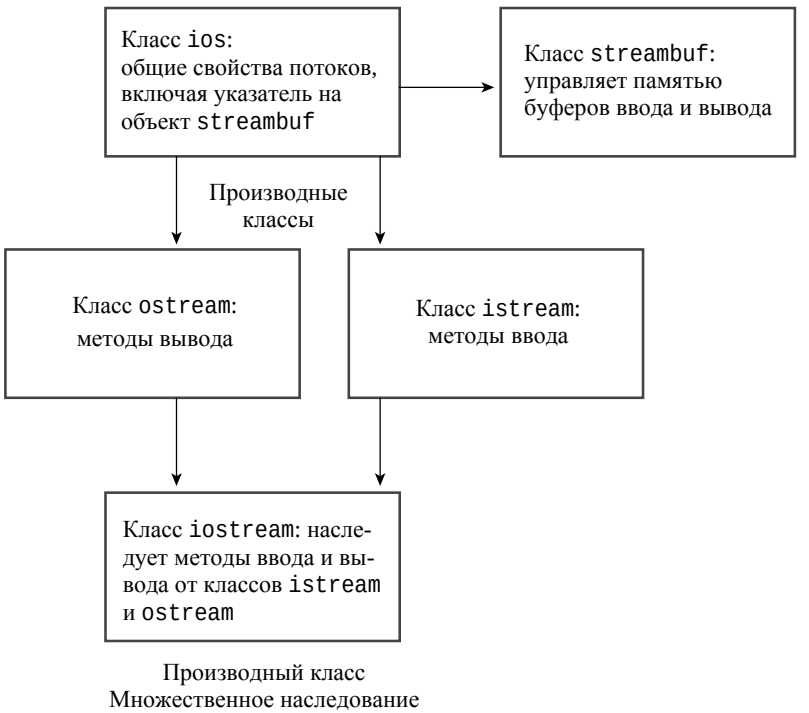


***Рис. 17.3.** Некоторые классы ввода-вывода*

Чтобы воспользоваться этими средствами, нужны объекты соответствующих классов. Например, для обработки вывода применяется такой объект `ostream`, как `cout`. Создание этого объекта открывает поток, автоматически создает буфер и ассоциирует его с потоком. Это также делает доступными функции-члены класса.

Переопределение ввода–вывода

В стандарте ISO/ANSI C++98 ввод-вывод претерпел ряд изменений. Во-первых, файл `ostream.h` заменен файлом `ostream`, причем `ostream` помещает классы в пространство имен `std`. Во-вторых, классы ввода-вывода были переписаны. Чтобы быть интернациональным языком, C++ должен уметь обрабатывать интернациональные наборы символов, которые требуют использования символьного типа шириной в 16 бит или более. Поэтому традиционный 8-битный ("узкий") тип `char` был дополнен символьным типом `wchar_t` (или "широким"). Версия C++11 добавляет типы `char16_t` и `char32_t`. Каждому типу требуются собственные средства ввода-вывода. Вместо того чтобы разрабатывать два (а теперь четыре) отдельных набора классов, комитет по стандартизации создал набор шаблонов классов ввода-вывода, включая `basic_istream< charT, traits<charT> >` и `basic_ostream< charT, traits<charT> >`. Шаблон `traits<charT>`, в свою очередь, представляет собой шаблонный класс, который определяет конкретные характеристики символьного типа, такие как способы сравнения на эквивалентность и значение EOF (end of file — конец файла). Стандарт C++11 предоставляет специализации `char` и `wchar_t` классов ввода-вывода. Например, `istream` и `ostream` — это определения `typedef` для специализаций `char`. Аналогично, `wistream` и `wostream` — специализации `wchar_t`. Так, например, существует объект `wcout` для потокового вывода "широких" символов. Заголовочный файл `ostream` содержит эти определения.

Определенная независимая от типа информация, которая хранилась в базовом классе `ios`, теперь перемещена в новый класс `ios_base`. Это относится к различным константам форматирования, таким как `ios::fixed`, которая теперь превратилась в `ios_base::fixed`. Класс `ios_base` содержит также некоторые опции, которых не было в старом `ios`.

Библиотека классов `iostream` в C++ берет на себя заботу о многих нюансах. Например, включение в программу файла `iostream` автоматически создает восемь потоковых объектов (четыре для потоков "узких" символов и четыре — для "широких").

- Объект `cin` соответствует стандартному потоку ввода. По умолчанию этот поток ассоциируется со стандартным устройством ввода — обычно клавиатурой. Объект `wcin` аналогичен ему, но работает с типом `wchar_t`.
- Объект `cout` соответствует стандартному потоку вывода. По умолчанию этот поток ассоциируется со стандартным устройством вывода — обычно монитором. Объект `wcout` аналогичен ему, но работает с символами типа `wchar_t`.
- Объект `cerr` соответствует стандартному потоку ошибок, который можно использовать для отображения сообщений об ошибках. По умолчанию этот поток ассоциируется со стандартным устройством вывода — обычно монитором — и данный поток не буферизуется. Это означает, что информация отправляется непосредственно на экран без ожидания заполнения буфера или передачи символа перевода строки. Объект `wcerr` аналогичен, но работает с типом `wchar_t`.
- Объект `clog` также соответствует стандартному потоку ошибок. По умолчанию этот поток ассоциируется со стандартным устройством вывода — обычно монитором — и не буферизуется. Объект `wclog` аналогичен, но работает с символами типа `wchar_t`.

Что означает утверждение, что объект представляет поток? Например, когда в файле `iostream` объявляется объект `cout` для программы, этот объект получает члены данных, содержащие информацию относительно вывода, такую как ширина поля для отображения данных, количество знаков после десятичной точки, основание системы счисления для отображения целочисленных данных и адрес объекта `streambuf`, описывающего буфер, который используется выходным потоком. Оператор, подобный показанному ниже, помещает символы из строки `"Bjarne free"` в управляемый объектом `cout` буфер посредством указанного объекта `streambuf`:

```
cout << "Bjarne free";
```

Класс `ostream` определяет функцию `operator<<()`, используемую в этом операторе, а также поддерживает данные-члены `cout` с множеством других методов класса наподобие тех, что обсуждаются в настоящей главе ниже. Более того, C++ учитывает, что вывод из буфера направляется в стандартный вывод — обычно монитор — предоставленный операционной системой. Короче говоря, с одной стороны поток подключен к программе, а с другой — к устройству стандартного вывода, и объект `cout` с помощью объекта типа `streambuf` управляет движением байтов в потоке.

Перенаправление

Стандартные потоки ввода и вывода обычно подключаются соответственно к клавиатуре и экрану. Но многие операционные системы, включая Unix, Linux и Windows, поддерживают перенаправление — функциональную возможность, которая позволяет заменять ассоциации стандартного ввода и стандартного вывода. Предположим, например, что имеется исполняемая программа командной строки `counter.exe`, напи-

санная на языке C++, которая подсчитывает количество символов ввода и выдает результат. (В большинстве версий Windows в меню Start (Пуск) можно выбрать пункт All Programs (Все программы) и затем щелкнуть на значке Command Prompt (Командная строка), чтобы открыть открытия окно командной строки.) Пример выполнения `counter.exe` может выглядеть следующим образом:

```
C>counter
Hello
and goodbye!
Control-Z                          ← эмуляция конца файла
Input contained 19 characters.
C>
```

В данном случае ввод поступает с клавиатуры, а вывод направляется на экран. Используя перенаправление ввода (<) и перенаправление вывода (>), эту же программу можно применить для подсчета количества символов в файле `oklahoma` и поместить результат в файл `cow_cnt`:

```
C>counter <oklahoma >cow_cnt
C>
```

Часть `<oklahoma` командной строки ассоциирует стандартный ввод с файлом `oklahoma`, что заставляет `cin` считывать ввод из этого файла вместо клавиатуры. Другими словами, операционная система заменяет подключение начальной части входного потока, в то время как конечная его часть остается подключенной к программе. Часть `>cow_cnt` командной строки ассоциирует стандартный вывод с файлом `cow_cnt`, что заставляет `cout` направлять свой вывод в этот файл, а не на экран. То есть операционная система заменяет конечную часть выходного потока, оставляя его начальную часть подключенной к программе. DOS, Windows в режиме командной строки, Linux и Unix автоматически распознают этот синтаксис перенаправления. (Все они, кроме ранних версий DOS, допускают использование необязательных символов пробела между операциями перенаправления и именами файлов.)

Стандартный выходной поток, представленный объектом `cout` – это нормальный канал вывода программы. Стандартные потоки ошибок (представленные объектами `cerr` и `clog`) предназначены для сообщений об ошибках программы. По умолчанию все эти три объекта отправляют информацию на монитор. Но перенаправление стандартного вывода не затрагивает `cerr` и `clog`; таким образом, если один из этих объектов используется для печати сообщений об ошибках, программа отобразит их на экране, даже если обычный вывод `cout` будет перенаправлен куда-либо в другое место. Например, рассмотрим следующий фрагмент кода:

```
if (success)
   std::cout << "Here come the goodies!\n";
else
{
   std::cerr << "Something horrible has happened.\n";
   exit(1);
}
```

Если перенаправление не используется, все сообщения будут выведены на экран. Однако если вывод программы перенаправлен в файл, то первое сообщение, если оно будет выбрано, будет направлено в файл, а второе сообщение в случае его выбора будет выведено на экран. Кстати, некоторые операционные системы допускают также перенаправление стандартного потока ошибок. Например, в Unix и Linux операция `2>` перенаправляет стандартный поток ошибок.

Вывод с помощью cout

Как упоминалось ранее, C++ рассматривает вывод как поток байтов. (В зависимости от реализации и платформы это могут быть 8-, 16- или 32-битные байты, но все равно они будут байтами.) Однако многие виды данных в программе организованы в виде более крупных блоков, нежели отдельный байт. Например, тип int может быть представлен 16- или 32-битным двоичным значением, а значение типа double — 64-битными двоичными данными. Но при отправке потока байтов на экран желательно, чтобы каждый байт представлял значение символа. То есть для отображения на экране числа -2.34 понадобится отправить пять символов: -, 2, ., 3 и 4, а не внутреннее 64-битное с плавающей точкой представление этого значения. Поэтому одной из наиболее важных задач, стоящих перед классом ostream, является преобразование числовых типов, таких как int или float, в поток символов, который представляет значения в текстовой форме. Таким образом, класс ostream транслирует внутреннее представление данных в виде двоичных битовых последовательностей в выходной поток символьных байтов. Для выполнения такой трансляции в классе ostream предусмотрено несколько методов. В этой главе мы рассмотрим их, резюмируя методы, которые используются на протяжении всей книги, и описывая дополнительные методы, обеспечивающие более тонкое управление выводом.

Перегруженная операция <<

Чаще всего в этой книге мы используем cout с операцией <<, которая также называется операцией *вставки*:

```
int clients = 22;
cout << clients;
```

В языке C++, как и в C, по умолчанию операция << используется в качестве битовой операции сдвига (см. приложение Д). Выражение наподобие x<<3 означает: загрузить двоичное представление x и сдвинуть все его биты на три позиции влево. Очевидно, что это не слишком тесно связано с выводом. Но класс ostream переопределяет операцию << для вывода. В этой ипостаси операция << называется операцией вставки, а не операцией сдвига влево. (Операция сдвига влево обретает эту новую роль посредством своего визуального аспекта, который предполагает смещение информации влево.) Операция вставки перегружена для применения со всеми базовыми типами C++:

- unsigned char
- signed char
- char
- short
- unsigned short
- int
- unsigned int
- long
- unsigned long
- long long (C++11)
- unsigned long long (C++11)
- float
- double
- long double

Класс ostream предоставляет определение функции operator<<() для каждого из этих типов данных. (Функции, имена которых содержат слово *operator*, используются для перегрузки операций, как описано в главе 11.) Таким образом, если применять оператор показанной ниже формы, и если *значение* относится к одному из перечисленных ранее типов, программа на языке C++ может сопоставить его с функцией операции с соответствующей сигнатурой:

```
cout << значение;
```

Например, выражение cout << 88 соответствует следующему прототипу метода:

```
ostream & operator<<(int);
```

Как вы должны помнить, такой прототип указывает, что функция operator<<() принимает один аргумент типа int. Это соответствует параметру 88 в предыдущем операторе. Прототип указывает также, что функция возвращает ссылку на объект ostream. Это свойство позволяет группировать вывод, как показано в следующем примере:

```
cout << "I'm feeling sedimental over " << boundary << "\n";
```

Если вы — программист на C и страдаете от многообразия спецификаторов типа % и проблем, связанных с несоответствием спецификатора действительному типу значения, то использование cout покажется вам до неприличия простым. (Разумеется, как и ввод C++ посредством cin.)

Вывод и указатели

Класс ostream определяет функции операции вставки для следующих типов указателей:

- const signed char *
- const unsigned char *
- const char *
- void *

Не забывайте, что C++ представляет строку, используя указатель на ее местоположение. Указатель может иметь форму имени массива элементов типа char, явного указателя на char или же строки в кавычках. Таким образом, все следующие операторы с cout отображают строки:

```
char name[20] = "Dudly Diddlemore";
char * pn = "Violet D'Amore";
cout << "Hello!";
cout << name;
cout << pn;
```

Для определения окончания последовательности отображаемых символов методы используют завершающий нулевой символ.

C++ интерпретирует указатель любого другого типа как void * и выводит числовое представление адреса. Если требуется получить адрес строки, необходимо выполнить приведение к другому типу, как показано в следующем фрагменте кода:

```
int eggs = 12;
char * amount = "dozen";
cout << &eggs;                  // выводит адрес переменной eggs
cout << amount;                 // выводит строку "dozen"
cout << (void *) amount;        // выводит адрес строки "dozen"
```

Конкатенация вывода

Все воплощения операции вставки определены с типом возвращаемого значения `ostream &`. То есть прототип имеет следующую форму:

```
ostream & operator<<(type);
```

(Здесь *type* — это тип отображаемого значения.) Возвращаемый тип `ostream &` означает, что использование этой операции возвращает ссылку на объект `ostream`. Но на какой объект? В соответствии с определением функции — это ссылка на тот же объект, который использован для вызова операции. Другими словами, функция операции возвращает тот же объект, который вызвал операцию. Например, `cout << "potluck"` возвращает объект `cout`. Это свойство позволяет выполнять конкатенацию вывода, используя вставку. Например, рассмотрим следующий оператор:

```
cout << "We have " << count << " unhatched chickens.\n";
```

Выражение `cout << "We have "` отображает строку и возвращает объект `cout`, сокращая оператор до следующего вида:

```
cout << count << " unhatched chickens.\n";
```

Затем выражение `cout << count` отображает значение переменной `count` и возвращает объект `cout`, который затем может использоваться для обработки заключительного аргумента в операторе (рис. 17.4). Это действительно изящная возможность, и именно поэтому в примерах перегрузки операции `<<` в предшествующих главах она имитировалась, как говорится, без зазрения совести.

Другие методы `ostream`

Помимо разнообразных функций `operator<<()`, класс `ostream` предоставляет метод `put()` для отображения символов и метод `write()` для отображения строк.

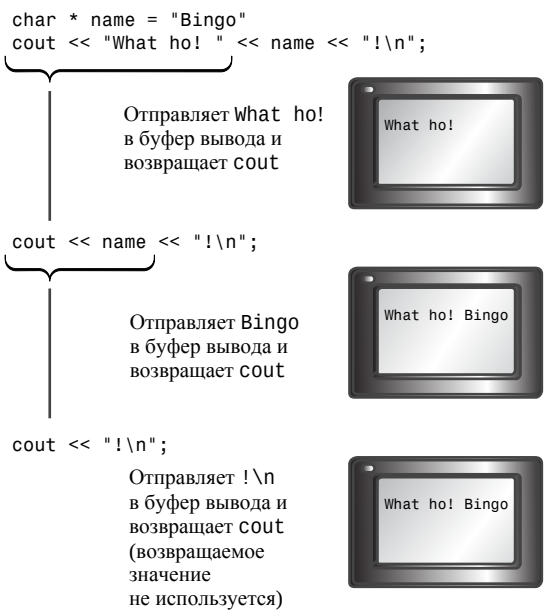


***Рис. 17.4.** Конкатенация вывода*

Изначально метод put() имел следующий прототип:

```
ostream & put(char);
```

Текущий стандарт эквивалентен первоначальному, за исключением того, что теперь метод реализован в виде шаблона, чтобы был возможен вывод значений типа wchar_t. Его вызов выполняется с использованием обычной нотации методов класса:

```
cout.put('W');                    // отображение символа W
```

Здесь cout — вызывающий объект, а put() — функция-член класса. Подобно функциям операции <<, эта функция возвращает ссылку на вызывающий объект, поэтому можно выполнить ее конкатенацию с выводом:

```
cout.put('I').put('t');       // отображение строки It с помощью
                              // двух вызовов функции put()
```

Вызов функции cout.put('I') возвращает объект cout, который затем служит вызывающим объектом для вызова функции put('t').

Располагая соответствующим прототипом, функцию put() можно применять с аргументами числовых типов, отличных от char, таких как int, и позволять прототипируемой функции автоматически преобразовывать аргумент к корректному значению типа char. Например, можно использовать следующие операторы:

```
cout.put(65);                     // отображает символ A
cout.put(66.3);                   // отображает символ B
```

Первый оператор преобразует целочисленное значение 65 в значение типа char и отображает символ, имеющий ASCII-код 65. Аналогично, второй оператор преобразует значение 66.3 типа double в значение 66 типа char и отображает соответствующий символ.

Такое поведение полезно при работе с версиями, предшествующими C++ Release 2.0. В этих версиях языка символьные константы представляются как значения типа int. Поэтому оператор, подобный показанному ниже, интерпретировал бы 'W' как значение типа int, и, следовательно, отобразил бы его в виде целочисленного значения 87 — ASCII-кода символа:

```
cout << 'W';
```

Но следующий оператор работает нормально:

```
cout.put('W');
```

Поскольку в текущей версии C++ константы char представляются типом char, можно использовать любой из описанных методов.

Некоторые компиляторы ошибочно перегружают put() для трех типов аргументов: char, unsigned char и signed char. Это приводит к неоднозначности при вызове put() с аргументом типа int, поскольку int может быть преобразован в любой из этих трех типов.

Метод write() записывает целую строку и имеет следующий шаблонный прототип:

```
basic_ostream<charT,traits>& write(const char_type* s, streamsize n);
```

Первый аргумент write() представляет адрес строки, которую нужно отобразить, а второй аргумент указывает количество отображаемых символов. Использование cout для вызова write() вызывает специализацию char, поэтому возвращаемым типом будет ostream &. В листинге 17.1 показано, как работает метод write().

Листинг 17.1. write.cpp

```
// write.cpp — использование cout.write()
#include <iostream>
#include <cstring>  // или иначе string.h

int main()
{
   using std::cout;
   using std::endl;
   const char * state1 = "Florida";
   const char * state2 = "Kansas";
   const char * state3 = "Euphoria";
   int len = std::strlen(state2);
   cout << "Increasing loop index:\n";
   int i;
   for (i = 1; i <= len; i++)
   {
      cout.write(state2,i);
      cout << endl;
   }

   // Конкатенация вывода
   cout << "Decreasing loop index:\n";
   for (i = len; i > 0; i--)
      cout.write(state2,i) << endl;

   // Превышение длины строки
   cout << "Exceeding string length:\n";
   cout.write(state2, len + 5) << endl;

   return 0;
}
```

Некоторые компиляторы могут обнаружить, что программа объявляет, но не использует массивы `state1` и `state3`. Все в порядке, поскольку эти два массива служат лишь для представления данных, находящихся в памяти перед и после массивом `state2`, чтобы можно было выяснить, что происходит в случае ошибки доступа к `state2`. Вывод программы из листинга 17.1 имеет следующий вид:

```
Increasing loop index:
K
Ka
Kan
Kans
Kansa
Kansas
Decreasing loop index:
Kansas
Kansa
Kans
Kan
Ka
K
Exceeding string length: Kansas Euph
```

Обратите внимание, что вызов `cout.write()` возвращает объект `cout`. Это обусловлено тем, что метод `write()` возвращает ссылку на объект, который его вызвал, и в данном случае это — объект `cout`.

Это позволяет выполнять конкатенацию вывода, поскольку `cout.write()` замещается возвращаемым значением — объектом `cout`:

```
cout.write(state2,i) << endl;
```

Вдобавок метод `write()` не прекращает вывод строки по достижении нулевого ограничивающего символа. Он просто выводит указанное количество символов, даже если при этом выходит за пределы конкретной строки! В данном случае программа объединяет строку `"Kansas"` с двумя другими стоками, как если бы соседние области памяти содержали единый элемент данных. Компиляторы могут по-разному размещать данные в памяти и по-разному выравнивать ее содержимое. Например, `"Kansas"` занимает 6 байтов, но, судя по всему, данный конкретный компилятор выравнивает строки, используя значения кратные 4 байтам, поэтому `"Kansas"` дополняется до 8 байт. Некоторые компиляторы разместят `"Florida"` после `"Kansas"`. Поэтому из-за различий в компиляторах последняя строка вывода может выглядеть по-разному.

Метод `write()` можно также использовать с числовыми данными. В этом случае ему нужно передать адрес числа, приведя его тип к `char *`:

```
long val = 560031841;
cout.write( (char *) &val, sizeof (long));
```

Это не ведет к трансляции числа в корректные символы; вместо этого такой вызов передает битовое представление хранящихся в памяти данных. Например, 4-байтное значение типа `long`, такое как `560031841`, будет передано в виде четырех отдельных байтов. Выходное устройство, подобное монитору, может затем попытаться интерпретировать каждый байт, как если бы он был ASCII-кодом (или каким-то другим). Поэтому число `560031841` отобразилось бы на экране в виде некоторой 4-символьной комбинации, скорее всего, бессмысленной (а может, и нет — проверьте и выясните). Однако `write()` обеспечивает компактный, аккуратный способ сохранения числовых данных в файле. Мы вернемся к этой возможности позднее в настоящей главе.

Очистка выходного буфера

Давайте посмотрим, что происходит, когда программа использует `cout` для отправки байтов в стандартный вывод. Поскольку буфер класса `ostream` управляется объектом `cout`, вывод не отправляется по назначению немедленно. Вместо этого он накапливается в буфере до тех пор, пока не заполнит его. Затем программа *очищает* буфер, отправляя его содержимое по назначению и освобождая буфер для приема новых данных. Как правило, размер буфера составляет 512 байт или же кратное этому значению число.

Буферизация — великолепный способ экономии времени, когда стандартный вывод подключен к файлу на жестком диске. В конце концов, мы же не хотим, чтобы программа обращалась к жесткому диску 512 раз для отправки 512 байт. Гораздо эффективнее накопить эти 512 байт в буфере и записать их на диск в ходе одной операции.

Однако для экранного вывода первоначальное заполнение буфера менее важно. В самом деле, было бы неудобно, если бы пришлось изменять сообщение типа "Нажмите любую клавишу для продолжения" так, чтобы оно заняло все 512 байт, предусмотренные для заполнения буфера. К счастью, в случае экранного вывода программе не обязательно дожидаться заполнения буфера. Например, отправка символа перевода строки обычно ведет к очистке буфера. Кроме того, как упоминалось ранее, большинство реализаций C++ очищают буфер, когда ожидается ввод.

Предположим, что имеется следующий код:

```
cout << "Enter a number: ";  // вывод приглашения на ввод числа
float num;
cin >> num;
```

Тот факт, что программа ожидает ввода, заставляет ее отобразить сообщение `cout` (т.е. очистить буфер от сообщения `"Enter a number: "`) немедленно, даже несмотря на то, что строка вывода не содержит символа новой строки. Не будь этой функциональной особенности, программе пришлось бы ожидать ввода, не выдав пользователю приглашения на ввод посредством сообщения `cout`.

Если используемая реализация не очищает буфер вывода, когда требуется, это можно сделать принудительно с помощью одного из двух манипуляторов. Манипулятор `flush` просто очищает буфер, а манипулятор `endl` очищает буфер и вставляет символ перевода строки. Эти манипуляторы применяются так же, как имена переменных:

```
cout << "Hello, good-looking! " << flush;
cout << "Wait just a moment, please." << endl;
```

Фактически манипуляторы являются функциями. Например, буфер `cout` можно очистить, вызвав функцию `flush()` непосредственно:

```
flush(cout);
```

Однако класс `ostream` перегружает операцию `<<` так, что следующее выражение замещается вызовом функции `flush(cout)`:

```
cout << flush
```

Таким образом, для очистки буфера можно с успехом использовать более удобную форму записи операции вставки.

Форматирование с помощью `cout`

Операции вставки в `ostream` преобразуют значения в текстовую форму. По умолчанию они форматируют значения следующим образом.

- Значение типа `char`, если оно представляет печатаемый символ, отображается как символ в поле шириной в один символ.
- Целочисленные типы отображаются как десятичные целые в поле с шириной, достаточной для отображения числа и знака минус (если таковой присутствует).
- Строки отображаются в поле ширины, равной длине строки.

Поведение по умолчанию для отображения чисел с плавающей точкой изменилось. Ниже описаны различия между старой и текущей реализациями C++.

- **Новый стиль.** Типы с плавающей точкой отображаются в поле общей шириной в шесть разрядов, за исключением завершающих нулей, которые не отображаются. (Следует отметить, что количество отображаемых разрядов никак не связано с точностью, с которой хранится число.) Число отображается либо в форме записи с фиксированной точкой, либо в экспоненциальной форме записи (см. главу 3), в зависимости от значения числа. В частности, экспоненциальная запись используется, если порядок равен `6` или больше либо равен `-5` или меньше. Опять-таки, ширина поля выбирается такой, чтобы вместить все цифры и знак минус, если он присутствует. Поведение по умолчанию соответствует использованию функции `fprintf()` из стандартной библиотеки C со спецификатором `%g`.

- **Старый стиль.** Типы с плавающей точкой отображаются в виде значений с шестью разрядами после десятичной точки, за исключением завершающих нулей, которые не отображаются. (Следует отметить, что количество отображаемых разрядов никак не связано с точностью, с которой хранится число.) Число отображается либо в форме записи с фиксированной точкой, либо в экспоненциальной форме записи (см. главу 3), в зависимости от значения числа. Опять-таки, ширина поля выбирается такой, чтобы вместить все цифры и знак минус, если он присутствует.

Поскольку каждое значение отображается в поле, ширина которого равна его размеру, необходимо побеспокоиться о явном указании пробелов между значениями. В противном случае соседние значения сольются.

Установки вывода по умолчанию иллюстрируются в листинге 17.2. Приведенный в нем код отображает двоеточие (`:`) после каждого значения, чтобы можно было видеть ширину поля, используемую в каждом случае. В программе применяется выражение `1.0 / 9.0` для генерации бесконечной дроби, чтобы продемонстрировать количество выводимых разрядов.

> **На заметку!**
>
> Не все компиляторы генерируют вывод, сформатированный в соответствии с текущим стандартом C++. К тому же текущий стандарт допускает региональные вариации. Например, европейская реализация может следовать континентальному стилю использования запятой вместо точки для отделения дробной части. То есть она может выводить `2,54` вместо `2.54`. Библиотека локализации (заголовочный файл `locale`) представляет механизм *оформления* входного или выходного потоков в определенном стиле, поэтому один компилятор может предоставлять более одного варианта выбора локальных настроек. В настоящей главе используются установки локализации для США.

Листинг 17.2. `defaults.cpp`

```
// defaults.cpp -- форматы по умолчанию cout
#include <iostream>
int main()
{
   using std::cout;
   cout << "12345678901234567890\n";
   char ch = 'K';
   int t = 273;
   cout << ch << ":\n";
   cout << t << ":\n";
   cout << -t <<":\n";

   double f1 = 1.200;
   cout << f1 << ":\n";
   cout << (f1 + 1.0 / 9.0) << ":\n";
   double f2 = 1.67E2;
   cout << f2 << ":\n";

   f2 += 1.0 / 9.0;
   cout << f2 << ":\n";
   cout << (f2 * 1.0e4) << ":\n";
   double f3 = 2.3e-4;
   cout << f3 << ":\n";
   cout << f3 / 10 << ":\n";
   return 0;
}
```

Ниже показан вывод программы из листинга 17.2:

```
12345678901234567890
K:
273:
-273:
1.2:
1.31111:
167:
167.111:
1.67111e+006:
0.00023:
2.3e-005:
```

Каждое значение заполняет свое поле. Обратите внимание, что завершающие нули в значении `1.200` не отображаются, но значения с плавающей точкой без завершающих нулей отображаются с шестью знаками. Кроме того, данная конкретная реализация отображает три разряда в экспоненте; другие могут использовать только два.

Изменение основания системы счисления, используемого для отображения

Класс `ostream` унаследован от класса `ios`, который, в свою очередь, унаследован от `ios_base`. Класс `ios_base` хранит информацию, описывающую состояние формата. Например, определенные биты в одном члене класса определяют используемое основание системы счисления, в то время как другие — ширину поля. Применяя *манипуляторы*, можно управлять основанием системы счисления, используемым для отображения целочисленных значений. С помощью функций-членов `ios_base` можно управлять шириной поля и количеством знаков после запятой. Поскольку класс `ios_base` является косвенным базовым классом для `ostream`, его методы можно применять с объектами класса `ostream` (или его наследников), например, с `cout`.

> **На заметку!**
>
> Члены и методы класса `ios_base` ранее находились в классе `ios`. Теперь же `ios_base` — это базовый класс для `ios`. В новой системе `ios` — шаблонный класс со специализациями `char` и `wchar_t`, а `ios_base` содержит нешаблонные средства.

Давайте посмотрим, как устанавливается основание системы счисления для отображения целых чисел. Для управления отображением целых чисел с использованием оснований 10, 16 или 8 можно применять манипуляторы `dec`, `hex` и `oct`. Например, следующий вызов функции устанавливает для объекта `cout` шестнадцатеричную систему счисления:

```
hex(cout);
```

После этого программа будет выводить целые значения в шестнадцатеричной форме до тех пор, пока не будет установлено другое основание. Обратите внимание, что манипуляторы не являются функциями-членами, поэтому не должны вызываться объектом.

Хотя в действительности манипуляторы являются функциями, обычно их используют следующим образом:

```
cout << hex;
```

Класс `ostream` перегружает операцию `<<` так, что она становится эквивалентной вызову функции `hex(cout)`. Манипуляторы определены в пространстве имен `std`. Применение этих манипуляторов иллюстрируется в листинге 17.3. Эта программа

отображает значение целого числа и его квадрата в трех разных системах счисления. Обратите внимание, что манипуляторы можно использовать отдельно или как часть последовательности вставок.

Листинг 17.3. manip.cpp

```
// manip.cpp -- использование манипуляторов формата
#include <iostream>
int main()
{
   using namespace std;
   // Вывод приглашения к вводу целого числа
   cout << "Enter an integer: ";

   int n;
   cin >> n;
   cout << "n    n*n\n";
   cout << n << "    " << n * n << " (decimal)\n";

   // Установка шестнадцатеричного режима вывода
   cout << hex;
   cout << n <<    " ";
   cout << n * n << " (hexadecimal)\n";

   // Установка восьмеричного режима
   cout << oct << n <<    " " << n * n << " (octal)\n";

   // Альтернативный способ вызова манипулятора
   dec(cout);
   cout << n << "    " << n * n << " (decimal)\n";

   return 0;
}
```

Ниже показан пример вывода программы из листинга 17.3:

```
Enter an integer: 13
n    n*n
13    169 (decimal)
d    a9 (hexadecimal)
15    251 (octal)
13    169 (decimal)
```

Настройка ширины полей

Возможно, вы заметили, что столбцы значений, выведенные программой из листинга 17.3, не выровнены. Это связано с тем, что числа имеют разную ширину полей. Для размещения чисел различной длины в полях постоянной ширины можно воспользоваться функцией-членом `width`. Этот метод имеет следующие прототипы:

```
int width();
int width(int i);
```

Первая форма возвращает текущую установку ширины поля. Вторая устанавливает ширину поля равной `i` пробелам и возвращает предыдущее значение ширины. Это позволяет сохранить предыдущее значение на случай, если позднее понадобится восстановить старое значение ширины поля.

Метод `width()` касается только следующего отображаемого элемента, после чего ширина поля возвращается к значению по умолчанию.

Например, рассмотрим следующие операторы:

```
cout << '#';
cout.width(12);
cout << 12 << "#" << 24 << "#\n";
```

Поскольку `width()` — функция-член, для ее вызова нужно указать объект (в данном случае — `cout`). Оператор вывода создает следующую строку вывода:

```
#          12#24#
```

Значение `12` помещается у правого края поля шириной в 12 символов. Это называется выравниванием по правому краю. После этого ширина поля возвращается в значение по умолчанию, и два символа `#` и значение `24` выводятся в полях, ширина которых равна длине этих элементов вывода.

Внимание!

Метод `width()` оказывает влияние только на следующий отображаемый элемент, после чего восстанавливается значение поля по умолчанию.

C++ никогда не усекает данные, поэтому при попытке вывода семизначного числа в двузначном поле C++ расширит это поле, чтобы вместить данные. (Некоторые языки заполняют поле звездочками, если ширина поля оказывается недостаточной для отображения данных. Разработчики C/C++ придерживаются подхода, в соответствии с которым отображение всех данных важнее сохранения аккуратного вида столбцов; C++ ставит на первое место сущность, а не форму.) Работа функции-члена `width()` продемонстрирована в листинге 17.4.

Листинг 17.4. `width.cpp`

```
// width.cpp -- использование метода width
#include <iostream>
int main()
{
   using std::cout;
   int w = cout.width(30);
   cout << "default field width = " << w << ":\n";  // ширина поля по умолчанию
   cout.width(5);
   cout << "N" <<':';
   cout.width(8);
   cout << "N * N" << ":\n";
   for (long i = 1; i <= 100; i *= 10)
   {
      cout.width(5);
      cout << i <<':';
      cout.width(8);
      cout << i * i << ":\n";
   }
   return 0;
}
```

Вывод программы из листинга 17.4 имеет следующий вид:

```
      default field width = 0:
  N:   N * N:
  1:       1:
 10:     100:
100:   10000:
```

Вывод отображает значения, выравнивая их по правому краю соответствующих полей. Поля вывода дополняются пробелами. То есть cout обеспечивает заданную ширину поля, добавляя пробелы. При выравнивании по правому краю пробелы вставляются слева от значений. Символ, используемый для дополнения поля, называется *символом-заполнителем*. По умолчанию устанавливается выравнивание по правому краю.

Обратите внимание, что программа в листинге 17.4 применяет ширину поля, равную 30, к строке, отображенной первым оператором cout, но не к значению w. Это объясняется тем, что метод width() оказывает влияние только на следующий одиночный отображаемый элемент. Обратите также внимание, что w имеет значение 0. Это обусловлено тем, что cout.width(30) возвращает предыдущую ширину поля, а не ширину, в которую оно только что было установлено. То, что значение w равно 0, означает, что нуль — значение поля по умолчанию. Поскольку C++ всегда расширяет поле, чтобы оно вместило данные, этот единственный размер подходит для всех возможных случаев. И, наконец, программа использует width() для выравнивания заголовков столбцов и данных, используя ширину равную пяти символам для первого столбца и равную восьми символам — для второго.

Символы-заполнители

По умолчанию cout заполняет неиспользуемые части поля пробелами. Для изменения этого можно воспользоваться функцией-членом fill(). Например, следующий вызов изменяет символ-заполнитель на звездочку:

```
cout.fill('*');
```

Это может быть удобно, например, для распечатки чеков, чтобы получатели не могли добавить к сумме один или два разряда. Применение этой функции-члена демонстрируется в листинге 17.5.

Листинг 17.5. fill.cpp

```
// fill.cpp -- изменение символа-заполнителя полей
#include <iostream>

int main()
{
   using std::cout;
   cout.fill('*');
   const char * staff[2] = { "Waldo Whipsnade", "Wilmarie Wooper"};
   long bonus[2] = {900, 1350};

   for (int i = 0; i < 2; i++)
   {
      cout << staff[i] << ": $";
      cout.width(7);
      cout << bonus[i] << "\n";
   }

   return 0;
}
```

Ниже показан вывод программы из листинга 17.5:

```
Waldo Whipsnade: $****900
Wilmarie Wooper: $***1350
```

Обратите внимание, что в отличие от ширины поля, новый символ-заполнитель остается в действии до тех пор, пока он не будет заменен.

Установка точности отображения чисел с плавающей точкой

Смысл *точности* чисел плавающей точкой зависит от режима вывода. В режиме по умолчанию она означает общее количество отображаемых разрядов. В фиксированной и научной нотации, которые мы обсудим позднее, *точность* означает количество десятичных разрядов, отображаемых справа от точки. Точность по умолчанию, применяемая в C++, как вы уже видели, равна 6. (Однако следует помнить, что завершающие нули отбрасываются.) Функция-член `precision()` позволяет выбрать другие значения. Например, следующий оператор устанавливает точность вывода `cout` в 2:

```
cout.precision(2);
```

В отличие от `width()`, но подобно `fill()`, новое значение точности остается в действии до тех пор, пока не будет переустановлено. Это демонстрируется в листинге 17.6.

Листинг 17.6. `precise.cpp`

```
// precise.cpp -- установка точности
#include <iostream>

int main()
{
   using std::cout;
   float price1 = 20.40;
   float price2 = 1.9 + 8.0 / 9.0;

   cout << "\"Furry Friends\" is $" << price1 << "!\n";
   cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

   cout.precision(2);
   cout << "\"Furry Friends\" is $" << price1 << "!\n";
   cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

   return 0;
}
```

Вывод программы из листинга 17.6 показан ниже:

```
"Furry Friends" is $20.4!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20!
"Fiery Fiends" is $2.8!
```

Обратите внимание, что в третьей строке этого вывода отсутствует завершающая десятичная точка. К тому же, в четвертой строке отображаются всего два разряда.

Вывод завершающих нулей и десятичных точек

Некоторые формы вывода, такие как цены или числа в столбцах, смотрятся лучше, если в них присутствуют завершающие нули. Например, вывод листинга 17.6 выглядел бы лучше, если бы сумма была представлена как `$20.40`, а не `$20.4`. В семействе классов `iostream` не предусмотрена функция, единственным назначением которой было бы обеспечение этого. Однако класс `ios_base` предоставляет функцию `setf()` (т.е. *set flag* – установить флаг), управляющую некоторыми средствами форматирования. Этот класс также определяет несколько констант, которые могут использоваться в качестве аргументов этой функции. Например, следующий вызов функции вынуждает `cout` отображать завершающие десятичные точки:

```
cout.setf(ios_base::showpoint);
```

В формате с плавающей точкой, используемом по умолчанию, это также обеспечивает отображение завершающих нулей. То есть вместо отображения значения 2.00 в виде 2, cout будет его отображать как 2.00000, если по умолчанию действует точность равная 6. В листинге 17.7 добавлен этот оператор.

На случай, если вас удивит форма записи ios_base::showpoint, следует отметить, что showpoint — это статическая константа с областью видимости класса, определенная в объявлении класса ios_base. Область видимости класса означает, что с именем константы нужно использовать операцию разрешения контекста (::), если это имя применяется вне определений функций-членов. Поэтому ios_base::showpoint именует константу, определенную в классе ios_base.

Листинг 17.7. showpt.cpp

```
// showpt.cpp -- установка точности с показом завершающей точки
#include <iostream>

int main()
{
    using std::cout;
    using std::ios_base;

    float price1 = 20.40;
    float price2 = 1.9 + 8.0 / 9.0;

    cout.setf(ios_base::showpoint);
    cout << "\"Furry Friends\" is $" << price1 << "!\n";
    cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

    cout.precision(2);
    cout << "\"Furry Friends\" is $" << price1 << "!\n";
    cout << "\"Fiery Fiends\" is $" << price2 << "!\n";

    return 0;
}
```

Вывод программы из листинга 17.7 с применением текущего форматирования C++ имеет следующий вид:

```
"Furry Friends" is $20.4000!
"Fiery Fiends" is $2.78889!
"Furry Friends" is $20.!
"Fiery Fiends" is $2.8!
```

Первая строка этого вывода содержит завершающие нули. Третья строка содержит десятичную точку, но в ней отсутствуют завершающие нули, поскольку точность установлена в 2, а два разряда уже отображены.

Дополнительные сведения о setf()

Метод setf() управляет еще несколькими установками формата помимо того, когда должна отображаться десятичная точка; рассмотрим их подробнее.

Класс ios_base имеет защищенный член данных, в котором отдельные биты (в данном контексте — *флаги*) управляют различными аспектами форматирования, такими как основание системы счисления и признак необходимости отображения завершающих нулей. Включение флага называется *установкой флага* (или бита) и означает установку его значения в 1. (Битовые флаги — это программный эквивалент установки двухрядных (DIP — dual-in-line package) переключателей при конфигурировании компьютерного оборудования.) Например, манипуляторы hex, dec и oct настраивают три битовых флага, управляющих выбором основания системы счисления.

Функция setf() предоставляет дополнительные средства настройки битов флагов. Функция setf() имеет два прототипа. Первый из них выглядит следующим образом:

```
fmtflags setf(fmtflags);
```

Здесь fmtflags — заданное посредством typedef имя типа *bitmask* (см. ниже врезку "На заметку!"), применяемое для хранения флагов форматирования. Имя определено в классе ios_base. Данная версия setf() используется для установки информации о формате, управляемой единственным битом. Аргумент — это значение fmtflags, указывающее, какие биты следует установить. Возвращаемое значение — число типа fmtflags, отражающее предыдущие значения всех флагов. На случай, если позднее требуется восстановить исходные настройки всех флагов, это значение можно сохранить. Какое значение передается функции setf()? Если бит под номером 11 требуется установить в 1, нужно передать число, бит 11 которого установлен в 1. Бит с номером 11 возвращаемого значения будет иметь старое значение этого бита. Отслеживание битов выглядит (и действительно является) утомительной задачей. Однако вы не обязаны выполнять эту работу. Класс ios_base определяет константы, которые представляют значения битов. Некоторые из этих определений представлены в табл. 17.1.

Таблица 17.1. Константы форматирования

Константа	Значение
ios_base::boolalpha	Ввод и вывод значений типа bool, таких как true и false
ios_base::showbase	Использование при выводе префиксов основания C++ (0,0x)
ios_base::showpoint	Отображение завершающей десятичной точки
ios_base::uppercase	Использование прописных букв для шестнадцатеричного вывода и экспоненциальной записи
ios_base::showpos	Использование символа + перед положительными числами

На заметку!

Тип *bitmask* (битовая маска) предназначен для хранения значений индивидуальных битов. Он может быть целочисленным, перечислением enum или контейнером bitset из STL. Главная идея в том, что каждый бит доступен индивидуально и имеет собственный смысл. Пакет iostream использует типы битовых масок для хранения информации о состоянии потока.

Поскольку эти константы форматирования определены в классе ios_base, с ними должна применяться операция разрешения контекста. То есть нужно использовать ios_base::uppercase, а не просто uppercase. В отсутствие директивы using или объявления using потребуется применять операцию разрешения контекста, указывая, что эти имена принадлежат пространству имен std, т.е. std::ios_base::showpos и т.д. Изменения остаются в силе до тех пор, пока не будут переопределены. Применение некоторых из этих констант демонстрируется в листинге 17.8.

Листинг 17.8. setf.cpp

```
// setf.cpp — использование setf() для управления форматированием
#include <iostream>

int main()
{
   using std::cout;
   using std::endl;
```

```
    using std::ios_base;

    int temperature = 63;

    cout << "Today's water temperature: ";
    cout.setf(ios_base::showpos);              // показывать знак плюс
    cout << temperature << endl;

    cout << "For our programming friends, that's\n";
    cout << std::hex << temperature << endl;   // использование шестнадцатеричного
                                               // представления
    cout.setf(ios_base::uppercase);            // применение прописных символов в
                                               // шестнадцатеричном представлении
    cout.setf(ios_base::showbase);             // использование префикса 0X для
                                               // шестнадцатеричных значений
    cout << "or\n";
    cout << temperature << endl;
    cout << "How " << true << "! oops -- How ";
    cout.setf(ios_base::boolalpha);
    cout << true << "!\n";

    return 0;
}
```

Ниже показан вывод программы из листинга 17.8:

```
Today's water temperature: +63
For our programming friends, that's
3f
or
0X3F
How 0X1! oops -- How true!
```

Обратите внимание, что знак "плюс" используется только с версией десятичного представления. C++ обрабатывает шестнадцатеричные и восьмеричные значения как значения без знака; поэтому для них никакой знак не требуется. (Однако некоторые реализации C++ могут все же отображать знак плюса.)

Второй прототип `setf()` принимает два аргумента и возвращает предыдущие установки:

```
fmtflags setf(fmtflags , fmtflags );
```

Эта перегруженная форма функции используется для форматирования нюансов отображения, которые управляются более чем одним битом. Первый аргумент, как и ранее – значение `fmtflags`, которое содержит требуемые установки. Второй аргумент – это значение, которое очищает соответствующие биты. Например, предположим, что установка третьего бита в `1` означает использование основания `10`, установка четвертого бита – в `1` означает основание `8`, а установка пятого бита – основание `16`.

Предположим, что вывод выполняется с основанием 10, и его требуется переключить к основанию 16. Для этого нужно не только установить 5-й бит в `1`, но и 3-й – в `0` (эту операцию называют *очисткой бита*). Интеллектуальный манипулятор `hex` решает обе проблемы автоматически. Применение функции `setf()` требует несколько больших усилий, поскольку нужно использовать второй аргумент для указания очищаемых битов, а затем первый аргумент – для указания устанавливаемых битов. Эта проблема не настолько сложна, как может показаться, поскольку для этой цели в классе `ios_base` определены специальные константы (перечисленные в табл. 17.2). В частности, изменяя основание системы счисления, нужно применять константу `ios_base::basefield` в качестве второго аргумента, и `ios_base::hex` – в качестве первого аргумента.

Таким образом, следующий вызов функции оказывает то же действие, что и применение манипулятора hex:

```
cout.setf(ios_base::hex, ios_base::basefield);
```

Таблица 17.2. Аргументы функции setf(long, long)

Второй аргумент	Первый аргумент	Значение
ios_base::basefield	ios_base::dec	Использовать основание 10
	ios_base::oct	Использовать основание 8
	ios_base::hex	Использовать основание 16
ios_base::floatfield	ios_base::fixed	Использовать форму записи с фиксированной точкой
	ios_base::scientific	Использовать научную форму записи
ios_base::adjustfield	ios_base::left	Использовать выравнивание по левому краю
	ios_base::right	Использовать выравнивание по правому краю
	ios_base::internal	Выровненный по левому краю знак или префикс основания системы счисления, выровненное по правому краю значение

Класс ios_base определяет три набора флагов форматирования, которыми можно управлять описанным выше способом. Каждый набор состоит из одной константы, которая может использоваться в качестве второго аргумента, и двух или трех констант, применяемых в качестве первого аргумента. Второй аргумент очищает пакет связанных битов, а затем первый аргумент устанавливает один из этих битов в значение 1. В табл. 17.2 перечислены имена констант, используемых для второго аргумента setf(), ассоциированный с ними выбор констант для первого аргумента, а также их значения. Например, чтобы выбрать выравнивание по левому краю, необходимо применить ios_base::adjustfield для второго аргумента и ios_base::left – для первого. Выравнивание по левому краю означает начало вывода значения у левой границы поля, а выравнивание по правому краю – завершение вывода значения на правой границе поля. Внутреннее выравнивание означает размещение знака и префикса основания у левой границы поля, а самого значения – у его правой границы (к сожалению, C++ не предоставляет режима автоматического выравнивания).

Форма записи с фиксированной точкой означает применение стиля 123.4 для значений с плавающей точкой, независимо от размера числа. Аналогично научная форма записи означает применение стиля 1.23e04, независимо от величины числа. Тем, кто знаком со спецификаторами функции printf() из C, в понимании режимов может помочь то, что режим, используемый C++ по умолчанию, соответствует спецификатору %g, режим fixed – спецификатору %f, а scientific – спецификатору %e.

В соответствии со стандартом C++, как фиксированная, так и научная форма записи обладают следующими двумя свойствами.

- *Точность* означает количество десятичных разрядов справа от десятичного разделителя, а не общее число разрядов.
- Завершающие нули отображаются.

Функция `setf()` — это функция-член класса `ios_base`. Поскольку это — базовый класс для класса `ostream`, функцию можно вызывать, используя объект `cout`. Например, чтобы затребовать выравнивание по левому краю, можно использовать следующий вызов:

```
ios_base::fmtflags old = cout.setf(ios::left, ios::adjustfield);
```

А для восстановления предыдущих настроек должен применяться следующий оператор:

```
cout.setf(old, ios::adjustfield);
```

В листинге 17.9 приведены дополнительные примеры применения функции `setf()` с двумя аргументами.

На заметку!

Программа в листинге 17.9 использует математическую функцию, а некоторые системы C++ не выполняют автоматический поиск библиотеки математических функций. Например, некоторые системы Unix требуют применения следующей команды:

```
$ CC setf2.C -lm
```

Опция `-lm` указывает компоновщику, что следует искать математическую библиотеку. Аналогично, некоторые системы Linux, использующие `g++`, требуют использования этого же флага.

Листинг 17.9. `setf2.cpp`

```
// setf2.cpp -- использование setf() с двумя аргументами для управления
форматированием
#include <iostream>
#include <cmath>

int main()
{
   using namespace std;
   // Использовать выравнивание влево, показать знак плюс,
   // показать завершающие нули с точностью, равной 3
   cout.setf(ios_base::left, ios_base::adjustfield);
   cout.setf(ios_base::showpos);
   cout.setf(ios_base::showpoint);
   cout.precision(3);

   // Использовать экспоненциальную запись и сохранить старые установки формата
   ios_base::fmtflags old = cout.setf(ios_base::scientific,
      ios_base::floatfield);
   cout << "Left Justification:\n";
   long n;
   for (n = 1; n <= 41; n+= 10)
   {
      cout.width(4);
      cout << n << "|";
      cout.width(12);
      cout << sqrt(double(n)) << "|\n";
   }

   // Переключиться на внутреннее выравнивание
   cout.setf(ios_base::internal, ios_base::adjustfield);

   // Восстановить стиль отображения значений с плавающей точкой, заданный по умолчанию
   cout.setf(old, ios_base::floatfield);
```

```
    cout << "Internal Justification:\n";
    for (n = 1; n <= 41; n+= 10)
    {
       cout.width(4);
       cout << n << "|";
       cout.width(12);
       cout << sqrt(double(n)) << "|\n";
    }
    // Использовать выравнивание по правому краю и форму записи с фиксированной точкой
    cout.setf(ios_base::right, ios_base::adjustfield);
    cout.setf(ios_base::fixed, ios_base::floatfield);
    cout << "Right Justification:\n";
    for (n = 1; n <= 41; n+= 10)
    {
       cout.width(4);
       cout << n << "|";
       cout.width(12);
       cout << sqrt(double(n)) << "|\n";
    }
    return 0;
}
```

Вывод программы из листинга 17.9 имеет следующий вид:

```
Left Justification:
+1  |+1.000e+00  |
+11 |+3.317e+00  |
+21 |+4.583e+00  |
+31 |+5.568e+00  |
+41 |+6.403e+00  |
Internal Justification:
+  1|+       1.00|
+ 11|+       3.32|
+ 21|+       4.58|
+ 31|+       5.57|
+ 41|+       6.40|
Right Justification:
   +1|      +1.000|
  +11|      +3.317|
  +21|      +4.583|
  +31|      +5.568|
  +41|      +6.403|
```

Обратите внимание, как установка точности в 3 ведет к отображению всего трех разрядов при отображении по умолчанию чисел с плавающей точкой (используемом в этой программе для внутреннего выравнивания), в то время как фиксированный и научный режимы отображают три разряда справа от десятичной точки. (Количество разрядов, отображаемое в экспоненциальной части для экспоненциальной записи, зависит от реализации.)

Эффекты вызова функции `setf()` можно отменить с помощью функции `unsetf()`, имеющей следующий прототип:

```
void unsetf(fmtflags mask);
```

Здесь `mask` – битовый шаблон. Все биты `mask`, установленные в 1, вызывают сброс соответствующих битов. То есть `setf()` устанавливает биты в 1, а `unsetf()` возвращает биты к состоянию 0.

Например:

```
cout.setf(ios_base::showpoint);      // показать завершающую десятичную точку
cout.unsetf(ios_base::boolshowpoint); // не показывать завершающую
                                     // десятичную точку
cout.setf(ios_base::boolalpha);      // отображать true, false
cout.unsetf(ios_base::boolalpha);    // отображать 1, 0
```

Возможно, вы обратили внимание на то, что не существует специального флага для указания режима по умолчанию для отображения чисел с плавающей точкой. Система работает следующим образом. Форма записи с фиксированной точкой используется, если установлен бит фиксированного режима (и только этот бит). Научная форма записи используется, если установлен бит научного режима (и только этот бит). Любая другая комбинация, такая как отсутствие установленных битов или установка обоих битов, ведет к активизации режима по умолчанию. Поэтому одним из способов вызова режима по умолчанию является такой:

```
cout.setf(0, ios_base::floatfield); // перейти в режим по умолчанию
```

Второй аргумент выключает оба бита, а первый аргумент не устанавливает ни одного. Более короткий путь достижения того же эффекта — применение функции `unsetf()` с аргументом `ios_base::floatfield`:

```
cout.unsetf(ios_base::floatfield);  // перейти в режим по умолчанию
```

Если известно, что объект `cout` находился в режиме фиксированной точки, то в качестве аргумента `unsetf()` можно использовать `ios_base::fixed`, но применение `ios_base::floatfield` работает независимо от текущего состояния `cout`, поэтому это решение предпочтительнее.

Стандартные манипуляторы

Применение `setf()` — не слишком дружественный к пользователю подход к форматированию, поэтому C++ предлагает несколько манипуляторов, которые самостоятельно вызывают функцию `setf()`, автоматически передавая ей правильные аргументы. Мы уже встречали `dec`, `hex` и `oct`. Эти манипуляторы, большинство которых не были доступны в старых реализациях C++, работают подобно `hex`. Например, следующий оператор включает режимы выравнивания по левому краю и отображения с фиксированной десятичной точкой:

```
cout << left << fixed;
```

В табл. 17.3 описаны как упомянутые, так и другие манипуляторы.

Таблица 17.3. Некоторые стандартные манипуляторы

Манипулятор	Вызовы
`boolalpha`	`setf(ios_base::boolalpha)`
`noboolalpha`	`unset(ios_base::noboolalpha)`
`showbase`	`setf(ios_base::showbase)`
`noshowbase`	`unsetf(ios_base::showbase)`
`showpoint`	`setf(ios_base::showpoint)`
`noshowpoint`	`unsetf(ios_base::showpoint)`
`showpos`	`setf(ios_base::showpos)`
`noshowpos`	`unsetf(ios_base::showpos)`

Окончание табл. 17.3

Манипулятор	Вызовы
uppercase	setf(ios_base::uppercase)
nouppercase	unsetf(ios_base::uppercase)
internal	setf(ios_base::internal, ios_base::adjustfield)
left	setf(ios_base::left, ios_base::adjustfield)
right	setf(ios_base::right, ios_base::adjustfield)
dec	setf(ios_base::dec, ios_base::basefield)
hex	setf(ios_base::hex, ios_base::basefield)
oct	setf(ios_base::oct, ios_base::basefield)
fixed	setf(ios_base::fixed, ios_base::floatfield)
scientific	setf(ios_base::scientific, ios_base::floatfield)

Совет

Если ваша система поддерживает эти манипуляторы, воспользуйтесь их преимуществами. Если же нет, всегда можно применить `setf()`.

Заголовочный файл `iomanip`

При установке некоторых значения форматирования, таких как ширина поля, использовать средства `iostream` неудобно. Для упрощения задачи C++ предлагаст дополнительные манипуляторы в заголовочном файле `iomanip`. Они обеспечивают те же функциональные средства, что и рассмотренные выше, но более удобным образом. Три наиболее часто используемые из них функции — это `setprecision()` для установки точности, `setfill()` для установки символа-заполнителя и `setw()` для установки ширины поля. В отличие от ранее рассмотренных манипуляторов, эти принимают аргументы. Манипулятор `setprecision()` принимает целочисленный аргумент, задающий точность, `setfill()` принимает аргумент типа `char`, указывающий символ-заполнитель, а `setw()` принимает целочисленный аргумент, устанавливающий ширину поля. Так как все они являются манипуляторами, их можно объединять операцией конкатенации в операторе `cout`. Это делает манипулятор `setw()` особенно удобным при отображении нескольких столбцов значений. Код в листинге 17.10 иллюстрирует это, несколько раз изменяя ширину поля и символ-заполнитель при выводе одной строки. Он использует также некоторые из более новых стандартных манипуляторов.

На заметку!

Некоторые системы C++ не выполняют автоматический поиск библиотеки математических функций. Как упоминалось ранее, некоторые системы Unix требуют выполнения следующего оператора для доступа к библиотеке математических функций:

```
$ CC iomanip.C -lm
```

Листинг 17.10. `iomanip.cpp`

```
// iomanip.cpp -- использование манипуляторов из iomanip
// некоторые манипуляторы требуют явной компоновки с библиотекой математических
функций
#include <iostream>
#include <iomanip>
```

```
#include <cmath>

int main()
{
   using namespace std;
   // Использование новых стандартных манипуляторов
   cout << fixed << right;

   // Использование манипуляторов iomanip для извлечения
   // квадратного корня и корня четвертой степени
   cout << setw(6) << "N" << setw(14) << "square root"
      << setw(15) << "fourth root\n";

   double root;  // извлечение корня
   for (int n = 10; n <=100; n += 10)
   {
      root = sqrt(double(n));
      cout << setw(6) << setfill('.') << n << setfill(' ')
           << setw(12) << setprecision(3) << root
           << setw(14) << setprecision(4) << sqrt(root)
           << endl;
   }
   return 0;
}
```

Ниже показан вывод программы из листинга 17.10:

```
     N  square root   fourth root
....10      3.162       1.7783
....20      4.472       2.1147
....30      5.477       2.3403
....40      6.325       2.5149
....50      7.071       2.6591
....60      7.746       2.7832
....70      8.367       2.8925
....80      8.944       2.9907
....90      9.487       3.0801
...100     10.000       3.1623
```

Теперь можно вывести аккуратно выровненные столбцы. Применение манипулятора `fixed` ведет к отображению завершающих нулей.

Ввод с помощью `cin`

Теперь обратимся к вводу и передаче данных в программу. Объект `cin` представляет стандартный ввод в виде потока байтов. Обычно этот поток символов генерируется клавиатурой. При вводе последовательности символов `2011` объект `cin` извлекает эти символы из входного потока. Этот ввод может являться частью строки, значением типа `int`, типа `float` или какого-либо иного типа. Таким образом, извлечение символов из потока предполагает также преобразование типа. Объект `cin` на основании типа переменной, предназначенной для приема значения, должен применять свои методы для преобразования последовательности символов в значения соответствующего типа. Обычно `cin` используют следующим образом:

```
cin >> value_holder;
```

Здесь *`value_holder`* идентифицирует область памяти, в которую помещается ввод. Это может быть именем переменной, ссылкой, разыменованным указателем либо чле-

ном структуры или класса. То, как cin интерпретирует ввод, зависит от типа данных *value_holder*. Класс istream, определенный в заголовочном файле iostream, перегружает операцию извлечения >> для распознавания следующих базовых типов:

- signed char &
- unsigned char &
- char &
- short &
- unsigned short &
- int &
- unsigned int &
- long &
- unsigned long &
- long long & (C++11)
- unsigned long long & (C++11)
- float &
- double &
- long double &

Их называют *функциями форматированного ввода*, потому что они преобразуют входные данные в соответствии с переменной назначения.

Типичная функция операции имеет прототип следующего вида:

```
istream & operator>>(int &);
```

И аргумент, и возвращаемое значение являются ссылками. При наличии аргумента-ссылки (см. главу 8) оператор, подобный показанному ниже, вынуждает функцию operator>>() работать с самой переменной staff_size, а не с ее копией, как это имеет место при использовании обычного аргумента:

```
cin >> staff_size;
```

Поскольку тип аргумента является ссылкой, cin в состоянии непосредственно модифицировать значение переменной, применяемой в качестве аргумента.

Например, предыдущий оператор непосредственно модифицирует значение переменной staff_size. О важности использования ссылки в качестве возвращаемого значения мы поговорим немного позже. Сначала необходимо исследовать аспект преобразования типа операцией извлечения. Для обработки аргументов каждого типа из приведенного выше списка операция извлечения преобразует символьный ввод в значение указанного типа. Например, предположим, что staff_size имеет тип int. В этом случае компилятор сопоставляет

```
cin >> staff_size;
```

со следующим прототипом:

```
istream & operator>>(int &);
```

Затем функция, соответствующая прототипу, считывает поток символов, отправляемый программе — например, символы 2, 3, 1, 8 и 4. Для систем, использующих двухбайтный тип int, функция преобразует эти символы в двухбайтное двоичное пред-

ставление целочисленного значения 23184. С другой стороны, если бы переменная staff_size имела тип double, объект cin использовал бы operator>>(double &), чтобы преобразовать этот же ввод в восьмибайтное представление значения с плавающей точкой 23184.0.

Кстати, вместе с cin можно применять манипуляторы hex, oct и dec для указания того, что вводимое целое должно интерпретироваться в шестнадцатеричном, восьмеричном или десятичном формате. Например, следующий оператор приводит к тому, что ввод 12 или 0x12 воспринимается как шестнадцатеричное значение 12, или десятичное 18, а ff или FF считывается как десятичное 255:

```
cin >> hex;
```

Класс istream также перегружает операцию извлечения >> для типов указателей на символы:

- signed char *
- char *
- unsigned char *

Для аргументов этих типов операция извлечения считывает следующее слово из входного потока и помещает его по указанному адресу, добавляя нулевой символ для ограничения строки. Например, предположим, что имеется следующий код:

```
// Вывод приглашения на ввод имени
cout << "Enter your first name:\n";
char name[20];
// Ввод имени
cin >> name;
```

Если вы ответите на запрос вводом Liz, то операция извлечения поместит символы Liz\0 в массив name. (Как обычно, \0 представляет завершающий нулевой символ.) Идентификатор name, будучи именем массива типа char, действует как адрес первого элемента массива и имеет тип char * (указатель на массив типа char).

То, что каждая операция извлечения возвращает ссылку на вызывающий объект, позволяет выполнять конкатенацию ввода, подобно тому, как это делается в отношении вывода:

```
char name[20];
float fee;
int group;
cin >> name >> fee >> group;
```

Здесь, к примеру, объект cin, возвращенный операцией cin >> name, становится тем объектом, который принимает fee.

Восприятие ввода операцией cin >>

Различные версии операции извлечения одинаково воспринимают входной поток. Они опускают пробельные символы (пробелы, символы новой строки и табуляции) до тех пор, пока не встретят непробельный символ. Это справедливо даже для односимвольных режимов (когда аргумент имеет тип char, unsigned char или signed char), но не относится к функциям символьного ввода языка C (рис. 17.5). В односимвольных режимах операция >> считывает символ и присваивает его указанному целевому объекту. В других режимах она считывает один элемент данных указанного типа. То есть она читает все, начиная с начального символа, отличного от пробельного, вплоть до первого символа, не соответствующего целевому типу.

Например, рассмотрим следующий код:

```
int elevation;
cin >> elevation;
```

Предположим, вы вводите следующие символы:

```
-123Z
```

Операция прочитает символы `-`, `1`, `2` и `3`, потому что они являются допустимыми составляющими целого числа. Но символ `Z` таковым не является, поэтому последним принятым символом ввода будет `3`. Символ `Z` останется во входном потоке, и следующий оператор `cin` начнет чтение с этой позиции. Тем временем, операция преобразует последовательность символов `-123` в целое значение и присвоит его переменной `elevation`.

```
char philosophy[20];
int distance;
char initial;

cin >> philosophy >> distance >> initial;
```

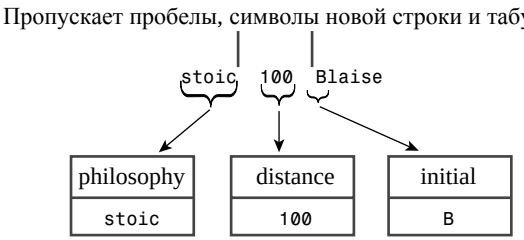


***Рис. 17.5.** Операция `cin >>` пропускает пробельные символы*

Может случиться, что ввод не оправдает ожиданий программы. Например, предположим, что вы ввели `Zcar` вместо `-123Z`. В этом случае операция извлечения оставит значение переменной `elevation` без изменений и возвратит значение `0`. (Точнее, оператор `if` или `while` оценивает объект `ifstream` как `false`, если установлено состояние ошибки; подробнее мы рассмотрим это позднее в этой главе.) Возвращаемое значение `false` позволит программе проверить, отвечает ли ввод требованиям программы, как показано в листинге 17.11.

Листинг 17.11. `check_it.cpp`

```
// check_it.cpp -- проверка допустимости ввода
#include <iostream>

int main()
{
   using namespace std;
   cout << "Enter numbers: ";  // запрос на ввод чисел

   int sum = 0;
   int input;
   while (cin >> input)
   {
      sum += input;
   }
```

```
    cout << "Last value entered = " << input << endl; // вывод последнего введенного значения
    cout << "Sum = " << sum << endl;                  // вывод суммы введенных чисел
    return 0;
}
```

Ниже показан вывод программы из листинга 17.11, когда во входной поток попадает неподходящее значение (-123Z):

```
Enter numbers: 200
10 -50 -123Z 60
Last value entered = -123
Sum = 37
```

Поскольку ввод буферизуется, вторая строка значений, введенных с клавиатуры, не посылается программе до тех пор, пока не будет нажата клавиша <Enter> в конце строки. Но цикл прекращает обработку на символе Z, потому что он не соответствует формату чисел с плавающей точкой. Несоответствие ввода ожидаемому формату, в свою очередь, приводит к тому, что выражение `cin >> input` оценивается как `false`, что прекращает выполнение цикла `while`.

Состояния потока

Внимательнее рассмотрим, что происходит при несоответствующем вводе. Объект `cin` или `cout` содержит член данных (унаследованный от класса `ios_base`), описывающий *состояние потока*. Состояние потока (определенное как тип `iostate`, представляющий собой описанный ранее тип битовой маски) состоит из трех элементов `ios_base`: `eofbit`, `badbit` и `failbit`. Каждый элемент – это отдельный бит, который может принимать значение `1` (установлен) или `0` (сброшен). Когда операция `cin` достигает конца файла, она устанавливает `eofbit` в `1`. Когда операция `cin` не может прочитать ожидаемые символы, как в предыдущем примере, она устанавливает `failbit` в `1`. Сбои ввода-вывода, вроде попытки чтения недоступного файла или попытки записи на защищенный от записи диск, также могут привести к установке `failbit` в `1`. Элемент `badbit` устанавливается в `1`, когда происходит некоторый не поддающийся диагностике сбой, который может повредить поток. (Конкретные реализации не всегда согласуются между собой в том, какие события должны устанавливать `failbit`, а какие – `badbit`.) Когда все эти три бита состояния установлены в `0`, все в порядке. Программы могут проверять состояния потока и использовать эту информацию, чтобы решить, что делать дальше. В табл. 17.4 перечислены эти биты наряду с методами `ios_base`, которые сообщают о состоянии либо изменяют его.

Таблица 17.4. Доступные состояния потоков

Член	Описание
`eofbit`	Устанавливается в `1` по достижении конца файла
`badbit`	Устанавливается в `1`, если поток может быть поврежден; например, в случае ошибки чтения файла
`failbit`	Устанавливается в `1`, если операция ввода не смогла прочитать ожидаемые символы или операция вывода не смогла их записать
`goodbit`	Просто другой способ выражения состояния `0`
`good()`	Возвращает `true`, если поток может быть использован (все биты очищены)

Окончание табл. 17.4

Член	Описание
`eof()`	Возвращает `true`, если бит `eofbit` установлен
`bad()`	Возвращает `true`, если бит `badbit` установлен
`fail()`	Возвращает `true`, если установлен бит `badbit` или `failbit`
`rdstate()`	Возвращает состояние потока
`exceptions()`	Возвращает битовую маску, идентифицирующую флаги, послужившие причиной исключения
`exceptions(iostate ex)`	Устанавливает состояния, которые будут вызывать `clear()` для генерации исключения; например, если `ex` — это `eofbit`, то `clear()` будет генерировать исключение, когда `eofbit` установлен
`clear(iostate s)`	Устанавливает состояние потока в `s`; по умолчанию значение `s` — `0` (`goodbit`); генерирует исключение `basic_ios::failure`, если `rdstate() & exceptions()) != 0`
`setstate(iostate s)`	Вызывает `clear(rdstate() | s)`. Это устанавливает биты состояния в соответствии с теми, что установлены в `s`; остальные биты состояния потока остаются неизменными

Установка состояния

Два метода в табл. 17.4 — `clear()` и `setstate()` — являются аналогичными. Они оба сбрасывают состояние, но делают это по-разному. Метод `clear()` устанавливает состояние в соответствии с переданным ему аргументом. Поэтому следующий вызов использует аргумент по умолчанию `0`, который очищает все три бита состояния (`eofbit`, `badbit` и `failbit`):

```
clear();
```

Подобным же образом следующий вызов делает состояние равным `eofbit`; т.е. бит `eofbit` устанавливается, а остальные два бита очищаются:

```
clear(eofbit);
```

Однако метод `setstate()` воздействует только на те биты, которые установлены в его аргументе. Таким образом, следующий вызов устанавливает `eofbit`, не затрагивая остальных битов:

```
setstate(eofbit);
```

Поэтому если `failbit` уже установлен, он таковым и останется.

Зачем может требоваться переустановка состояния потока? Для автора программ наиболее частая причина использования `clear()` без аргументов связана с необходимостью повторного открытия ввода после получения неподходящих данных или конца файла. То, имеет ли смысл делать это, зависит от стоящей перед программой цели. Скоро мы рассмотрим некоторые примеры. Основное назначение `setstate()` — предоставление средств изменения состояния функциям ввода и вывода. Например, если типом переменной `num` является `int`, следующий вызов может приводить к тому, что `operator>>(int &)` будет использовать `setstate()` для установки бита `failbit` или `eofbit`:

```
cin >> num;  // чтение int
```

Ввод-вывод и исключения

Предположим, что функция ввода установила `eofbit`. Приведет ли это к генерации исключения? По умолчанию нет. Однако для управления тем, как будут обрабатываться исключения, можно использовать метод `exceptions()`.

Сначала приведем некоторые основополагающие сведения. Метод `exceptions()` возвращает битовое поле с тремя битами, соответствующими `eofbit`, `failbit` и `badbit`. Изменение состояния потока ведет к применению либо метода `clear()`, либо метода `setstate()`, который использует `clear()`. После изменения состояния потока метод `clear()` сравнивает его текущее состояние со значением, возвращенным `exceptions()`. Если бит установлен в возвращаемом значении, и соответствующий бит установлен также в текущем состоянии, то `clear()` генерирует исключение `ios_base::failure`. Это может случиться, например, если оба значения имеют установленный бит `badbit`. Отсюда следует, что если `exceptions()` возвращает `goodbit`, то исключение не генерируется. Класс `ios_base::failure` унаследован от класса `std::exception`, а потому имеет метод `what()`.

Установкой по умолчанию для `exceptions()` является `goodbit` — т.е. никакие исключения не генерируются. Однако перегруженная функция `exceptions(iostate)` позволяет управлять этим поведением:

```
cin.exceptions(badbit);  // установка badbit ведет к генерации исключения
```

Как описано в приложении Д, битовая операция "ИЛИ" (`|`) позволяет указывать более одного бита. Например, следующий оператор вызывает генерацию исключения при последующей установке `badbit` или `eofbit`:

```
cin.exceptions(badbit | eofbit);
```

В листинге 17.12 приведен модифицированный код из листинга 17.11, который генерирует и перехватывает исключение при установке бита `failbit`.

Листинг 17.12. `cinexcp.cpp`

```
// cinexcp.cpp -- cin, генерирующий исключения
#include <iostream>
#include <exception>
int main()
{
   using namespace std;
   // Установленный бит failbit вызовет генерацию исключения
   cin.exceptions(ios_base::failbit);
   cout << "Enter numbers: ";  // запрос на ввод чисел
   int sum = 0;
   int input;
   try {
      while (cin >> input)
      {
         sum += input;
      }
   } catch(ios_base::failure & bf)
   {
      cout << bf.what() << endl;
      cout << "O! the horror!\n";
   }
   cout << "Last value entered = " << input << endl; // вывод последнего введенного числа
   cout << "Sum = " << sum << endl;                  // вывод суммы введенных чисел
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 17.12; сообщение, выдаваемое методом `what()`, зависит от реализации:

```
Enter numbers: 20 30 40 pi 6
ios_base failure in clear
O! the horror!
Last value entered = 40.00
Sum = 90.00
```

Подобным образом можно использовать исключения при вводе. Но должны ли вы их использовать? Это зависит от контекста. Для данного примера ответ будет отрицательным. Исключения должны перехватывать необычные, непредвиденные ситуации, но в этой конкретной программе несоответствие типов применяется в качестве способа выхода из цикла. Однако в этой программе было бы целесообразно генерировать исключение для установленного бита `badbit`, поскольку такая ситуация была бы непредусмотренной. Или же, если бы программа была предназначена для чтения данных из файла вплоть до его конца, тогда, возможно, имело бы смысл генерировать исключение для установленного бита `failbit`, поскольку он свидетельствовал бы о наличии проблемы с файлом данных.

Эффекты состояния потока

Проверка `if` или `while`, подобная показанной ниже, дает `true` только при нормальном состоянии потока (когда все биты очищены):

```
while (cin >> input)
```

Если проверка дает `false`, функции-члены, перечисленные в табл. 17.4, можно использовать для точного определения причины. Например, центральную часть листинга 17.11 можно модифицировать следующим образом:

```
while (cin >> input)
{
    sum += input;
}
if (cin.eof())
    cout << "Loop terminated because EOF encountered\n";
```

Установка битов состояния потока имеет очень важное следствие: поток закрывается для дальнейшего ввода или вывода до тех пор, пока бит не будет очищен. Например, следующий код не будет работать:

```
while (cin >> input)
{
    sum += input;
}
cout << "Last value entered = " << input << endl;
cout << "Sum = " << sum << endl;
cout << "Now enter a new number: ";
cin >> input;  // не будет работать
```

Если требуется, чтобы программа продолжала считывать ввод после установки бита состояния потока, его нужно сбросить в нормальное. Это можно сделать посредством вызова метода `clear()`:

```
while (cin >> input)
{
    sum += input;
}
```

```
cout << "Last value entered = " << input << endl;
cout << "Sum = " << sum << endl;
cout << "Now enter a new number: ";
cin.clear();          // сброс состояния потока
while (!isspace(cin.get()))
   continue;          // отбрасывание некорректного ввода
cin >> input;         // теперь будет работать
```

Обратите внимание, что сброса состояния потока недостаточно. Некорректный ввод, который привел к прекращению цикла ввода, по-прежнему остается во входной очереди, и программа должна его обработать. Один из способов достижения этого — продолжение считывания вплоть до пробельного символа. Функция `isspace()` (см. главу 6) — это функция `cctype`, которая возвращает `true`, если ее аргумент является пробельным символом. Или же можно отбросить остаток строки, а не только следующее слово:

```
while (cin.get() != '\n')
   continue;          // отбрасывание оставшейся части строки
```

В этом примере цикл прекращается по причине некорректного ввода. Но предположим, что цикл прекращается вследствие достижения конца файла либо из-за аппаратного сбоя. Тогда новый код, отбрасывающий некорректный ввод, лишен смысла. Ситуацию можно исправить, используя метод `fail()` для проверки корректности предположения. Поскольку по историческим причинам `fail()` возвращает `true`, если любой из битов `failbit` или `eofbit` установлен, код должен исключить последний случай. Следующий фрагмент демонстрирует пример такого исключения:

```
while (cin >> input)
{
   sum += input;
}
cout << "Last value entered = " << input << endl;
cout << "Sum = " << sum << endl;
if (cin.fail() && !cin.eof() )      // отрицательный результат из-за
                                    // некорректного ввода
{
   cin.clear();                     // сброс состояния потока
   while (!isspace(cin.get()))
      continue;                     // отбрасывание некорректного ввода
}
else                                // иначе требуется помощь
{
   cout << "I cannot go on!\n";     // вывод сообщения о невозможности
                                    // продолжения работы
   exit(1);
}
cout << "Now enter a new number: "; // запрос на ввод нового числа
cin >> input;                       // теперь будет работать
```

Другие методы класса `istream`

В главах с 3 по 5 рассматривались методы `get()` и `getline()`. Вспомните, что они обеспечивают следующие дополнительные возможности ввода.

- Методы `get(char &)` и `get(void)` обеспечивают односимвольный ввод без пропуска пробельных символов.

- Функции `get(char *, int, char)` и `getline(char *, int, char)` по умолчанию считывают строки целиком, а не отдельные слова.

Эти функции называются *функциями неформатированного ввода*, потому что они просто считывают символьный ввод, как он есть, не пропуская пробелы, переводы строк и символы табуляции и не выполняя никаких преобразований данных. Рассмотрим эти две группы функций-членов класса `istream`.

Односимвольный ввод

Когда метод `get()` вызываются с аргументом типа `char` или вообще без аргументов, он извлекает следующий символ ввода, даже если это пробел, знак табуляции или символ новой строки. Версия `get(char & ch)` присваивает введенный символ своему аргументу, а версия `get(void)` просто использует введенный символ, преобразованный в целочисленный тип (обычно — `int`), в качестве своего возвращаемого значения.

Функция-член `get(char &)`

Вначале рассмотрим `get(char &)`. Предположим, что в программе присутствует следующий цикл:

```
int ct = 0;
char ch;
cin.get(ch);
while (ch != '\n')
{
   cout << ch;
   ct++;
   cin.get(ch);
}
cout << ct << endl;
```

Далее предположим, что мы вводим следующую фразу:

```
I C++ clearly.<Enter>
```

Нажатие клавиши <Enter> отправляет эту строку ввода программе. Фрагмент программы считывает символ `I`, отображает его с помощью оператора `cout` и увеличивает значение `ct` до `1`. Затем он считывает символ пробела, следующий за `I`, отображает его и увеличивает значение `ct` до `2`. Это продолжается до тех пор, пока программа не обработает нажатие клавиши <Enter> как символ новой строки и не прекратит цикл. Главная идея в том, что с использованием `get(ch)` код читает, отображает и подсчитывает как пробелы, так и печатные символы.

Предположим теперь, что вместо этого программа использует операцию >>:

```
int ct = 0;
char ch;
cin >> ch;
while (ch != '\n')   // ОТРИЦАТЕЛЬНЫЙ РЕЗУЛЬТАТ ПРОВЕРКИ
{
   cout << ch;
   ct++;
   cin >> ch;
}
cout << ct << endl;
```

Прежде всего, этот код будет пропускать пробелы, соответственно не подсчитывая их и сжимая вывод:

```
IC++clearly.
```

Хуже того, цикл никогда не завершится! Поскольку операция извлечения пропускает символы новой строки, этот код никогда не присвоит такой символ переменной ch и, следовательно, проверка условия цикла while никогда не завершит его выполнения.

Функция-член get(char &) возвращает ссылку на объект istream, применяемый для ее вызова. Это значит, что можно выполнить конкатенацию извлечений, следующих за get(char &):

```
char c1, c2, c3;
cin.get(c1).get(c2) >> c3;
```

Для начала, cin.get(c1) присваивает первый введенный символ переменной c1 и возвращает вызывающий объект, в данном случае – cin. Это сокращает код до cin.get(c2) >> c3, который присваивает второй введенный символ переменной c2. Вызов функции возвращает cin, сокращая код до cin >> c3. Это, в свою очередь, присваивает переменной c3 следующий отличный от пробельного символ. Обратите внимание, что переменным c1 и c2 могут быть присвоены пробельные символы, но c3 – нет.

Если метод cin.get(char &) встречает конец файла – реальный или имитированный с клавиатуры (сочетанием клавиш <Ctrl+Z> в DOS и в режиме командной строки Windows, либо <Ctrl+D> в начале строки в системе Unix), он не присваивает значение своему аргументу. И это достаточно правильно, потому что если программа достигла конца файла, никакого значения для присваивания не существует. Более того, метод вызывает setstate(failbit), что приводит к возврату значения false в результате проверки cin:

```
char ch;
while (cin.get(ch))
{
    // обработка ввода
}
```

До тех пор, пока ввод корректен, возвращаемым значением cin.get(ch) остается объект cin, проверка которого возвращает значение true, поэтому цикл продолжает работать. При достижении конца файла возвращаемое значение вычисляется как false, что прекращает цикл.

Функция-член getchar()

Функция-член get(void) также считывает пробельные символы, но использует свое возвращаемое значение для передачи ввода в программу. Поэтому ее можно использовать следующим образом:

```
int ct = 0;
char ch;
ch = cin.get();    // использование возвращаемого значения
while (ch != '\n')
{
    cout << ch;
    ct++;
    ch = cin.get();
}
cout << ct << endl;
```

Функция-член get(void) возвращает тип int (или какой-то более длинный целочисленный тип, что зависит от набора символов и региональных установок).

Это делает следующий фрагмент неправильным:

```
char c1, c2, c3;
cin.get().get() >> c3;    // не допускается
```

В этом примере `cin.get()` возвращает значение типа `int`. Поскольку это возвращаемое значение не является объектом класса, к нему нельзя примснить операцию, предусмотренную только для членов класса. Таким образом, мы получаем синтаксическую ошибку. Однако функцию `get()` можно использовать в конце последовательности извлечений:

```
char c1;
cin.get(c1).get();        // допустимо
```

То, что `get(void)` возвращает тип `int`, означает, что вслед за ней нельзя вызвать операцию извлечения. Но поскольку выражение `cin.get(c1)` возвращает `cin`, оно становится подходящим префиксом для `get()`. Этот конкретный код прочитает первый символ ввода, присвоит его переменной `c1`, затем прочитает второй введенный символ и отбросит его.

По достижении конца файла — реального или эмулируемого — `cin.get(void)` возвращает значение `EOF`, которое является символьной константой, определенной в заголовочном файле `iostream`. Эта архитектурная особенность делает возможным использование следующей конструкции для считывания ввода:

```
int ch;
while ((ch = cin.get()) != EOF)
{
    // обработка ввода
}
```

В данном случае для переменной `ch` нужно использовать тип `int` вместо `char`, поскольку значение `EOF` не может быть выражено как тип `char`.

Несколько подробнее эти функции описаны в главе 5, а особенности функций односимвольного ввода кратко описаны в табл. 17.5.

Таблица 17.5. Сравнение `cin.get(ch)` и `cin.get()`

Свойство	`cin.get(ch)`	`ch = cin.get()`
Метод передачи входного символа	Присваивает аргументу переменную `ch`	Использует возвращаемое значение для присваивания переменной `ch`
Возвращаемое значение для символьного ввода	Ссылка на объект класса `istream`	Код символа как значение типа `int`
Возвращаемое значение функции	Преобразуется в `false`	`EOF` в конце файла

Какую форму односимвольного ввода предпочесть?

Что следует использовать при наличии выбора между `>>`, `get(char &)` и `get(void)`? Сначала нужно решить, требуется ли пропускать пробелы при вводе. Если пропуск пробелов допустим (или даже удобен), нужно использовать операцию извлечения `>>`. Например, пропуск пробелов удобен при реализации простого меню:

```
cout << "a. annoy client        b. bill client\n"
     << "c. calm client         d. deceive client\n"
     << "q.\n";
cout << "Enter a, b, c, d, or q: ";
```

```
char ch;
cin >> ch;
while (ch != 'q')
{
   switch(ch)
   {
      ...
   }
   cout << "Enter a, b, c, d, or q: ";
   cin >> ch;
}
```

Чтобы ввести, скажем, ответ b, вы вводите **b** и нажимаете клавишу <Enter>, генерируя двухсимвольный ответ b\n. Если использовать любую из форм get(), пришлось бы добавить код, обрабатывающий символ \n на каждом шаге цикла, но операция извлечения без труда пропускает его. (Если вы программировали на C, то, скорее всего, сталкивались с ситуацией, когда символ новой строки воспринимался программой как недопустимый ответ. Решение этой проблемы не представляет сложности, но все же создает некоторые неудобства.)

Если требуется, чтобы программа обрабатывала каждый символ, нужно применять один из методов get(). Например, программа подсчета символов может использовать пробельные символы в качестве признака конца слова. Из двух методов get() только get(char &) имеет интерфейс в виде класса. Главное преимущество метода get(void) в том, что он очень похож на стандартную функцию getchar() из C, а это значит, что программу на C можно преобразовать в C++, подключив заголовочный файл iostream вместо stdio.h и глобально заменив getchar() на cin.get(), а putchar(ch) — на cout.put(ch).

Строковый ввод: getline(), get() и ignore()

Теперь рассмотрим функции-члены строкового ввода, описанные в главе 4. И функция-член getline(), и строчно-ориентированная версия get() считывают строки, и обе они имеют одинаковую сигнатуру (здесь приведена упрощенная форма более общего объявления шаблона):

```
istream & get(char *, int, char);
istream & get(char *, int);
istream & getline(char *, int, char);
istream & getline(char *, int);
```

Вспомните, что первый аргумент является адресом помещения вводимой строки. Второй аргумент — значение на единицу больше максимального количества символов, которые следует прочитать. (Дополнительный символ обеспечивает место для ограничивающего нулевого символа, используемого при сохранении ввода в виде строки.) Третий аргумент указывает символ, служащий разделителем. Версии, имеющие только два аргумента, применяют в качестве разделителя символ перевода строки. Каждая функция считывает максимальное количество символов или все символы до тех пор, пока не встретит символ-разделитель — в зависимости от того, что случится раньше.

Например, следующий код считывает символьный ввод в символьный массив line:

```
char line[50];
cin.get(line, 50);
```

Функция cin.get() прекращает считывание ввода в массив после получения 49 символов или, по умолчанию, после получения символа перевода строки — в зависи-

мости от того, что произойдет раньше. Основное различие между get() и getline() в том, что get() оставляет символ перевода строки во входном потоке, делая его доступным для следующей операции ввода, в то время как getline() отбрасывает символы новой строки из входного потока.

Использование двухаргументной формы этих функций-членов было показано в главе 4. Теперь рассмотрим трехаргументные версии. Третий аргумент — это символ-разделитель. Появление разделителя прерывает считывание, даже если максимальное количество символов еще не прочитано. Поэтому по умолчанию оба метода прекращают считывание при достижении конца строки до чтения заданного количества символов. Как и в случае по умолчанию, get() оставляет символ-разделитель во входной очереди, а getline() — нет.

Код в листинге 17.13 демонстрирует работу getline() и get(). В нем также представлена функция-член ignore(). Функция ignore() принимает два аргумента: число, указывающее максимальное количество символов для чтения, и символ, служащий разделителем при вводе. Например, следующий вызов функции считывает и отбрасывает следующие 255 символов или все символы вплоть до символа перевода строки, в зависимости от того, что произойдет раньше:

```
cin.ignore(255, '\n');
```

Этот прототип предусматривает для своих двух аргументов значения по умолчанию 1 и EOF. Функция возвращает тип istream &:

```
istream & ignore(int = 1, int = EOF);
```

(Значение по умолчанию EOF вынуждает функцию ignore() считывать все символы вплоть до заданного количества или до конца файла, в зависимости от того, что случится раньше.)

Функция возвращает вызывающий объект. Это позволяет выполнять конкатенацию вызовов функции, как в следующем операторе:

```
cin.ignore(255, '\n').ignore(255, '\n');
```

В этом примере первый метод ignore() считывает и отбрасывает одну строку, а второй — вторую строку. Вместе эти две функции считывают две строки. Теперь взгляните на код в листинге 17.13.

Листинг 17.13. get_fun.cpp

```
// get_fun.cpp -- использование get() и getline()
#include <iostream>
const int Limit = 255;

int main()
{
    using std::cout;
    using std::cin;
    using std::endl;

    char input[Limit];

    cout << "Enter a string for getline() processing:\n";  // запрос на ввод строки
    cin.getline(input, Limit, '#');
    cout << "Here is your input:\n";
    cout << input << "\nDone with phase 1\n";

    char ch;
    cin.get(ch);
    cout << "The next input character is " << ch << endl;
```

```
    if (ch != '\n')
        cin.ignore(Limit, '\n');  // отбрасывание остальной части строки
    cout << "Enter a string for get() processing:\n";
    cin.get(input, Limit, '#');
    cout << "Here is your input:\n";
    cout << input << "\nDone with phase 2\n";
    cin.get(ch);
    cout << "The next input character is " << ch << endl;
    return 0;
}
```

Ниже приведен пример запуска программы из листинга 17.13:

```
Enter a string for getline() processing:
Please pass
me a #3 melon!
Here is your input:
Please pass
me a
Done with phase 1
The next input character is 3
Enter a string for get() processing:
I still
want my #3 melon!
Here is your input:
I still
want my
Done with phase 2
The next input character is #
```

Обратите внимание, что функция `getline()` отбрасывает ограничивающий символ # во вводе, а функция `get()` — нет.

Непредусмотренный строковый ввод

Некоторые формы ввода при использовании функций `get(char *, int)` и `getline()` влияют на состояние потока. Как и при использовании других функций ввода, достижение конца файла влечет за собой установку флага `eofbit`, а все, что повреждает поток — как, например, сбой устройства — приводит к установке `badbit`. Двумя другими особыми случаями является отсутствие ввода или превышение количества символов, указанного при вызове функции. Рассмотрим эти случаи.

Если любому из методов не удается извлечь никаких символов, во входную строку помещается нулевой символ и функция `setstate()` используется для установки `failbit`. А в каких случаях методу не удается извлечь никаких символов? Одной из причин может быть немедленное достижение конца файла. В случае применения функции `get(char *, int)` еще одной причиной может быть ввод пустой строки:

```
char temp[80];
while (cin.get(temp,80))  // завершение ввода на пустой строке
```

Интересно, что пустая строка не вынуждает `getline()` устанавливать флаг `failbit`. Это связано с тем, что `getline()` извлекает символ перевода строки, даже если и не сохраняет его. Если требуется, чтобы цикл `getline()` завершался на пустой строке, можно записать его следующим образом:

```
char temp[80];
while (cin.getline(temp,80) && temp[0] != '\0')  // завершение ввода
                                                 // на пустой строке
```

Теперь предположим, что количество символов во входной очереди соответствует или превышает максимальное, указанное методом ввода. Вначале рассмотрим `getline()` и следующий код:

```
char temp[30];
while (cin.getline(temp,30))
```

Метод `getline()` будет считывать последовательные символы из входной очереди, помещая их в последовательные элементы массива `temp` до тех пор, пока (в ходе проверки) не встретится конец файла, либо пока следующий символ не окажется символом новой строки, либо пока не будет сохранено 29 символов. Если достигается конец файла, устанавливается флаг `eofbit`. Если следующий предназначенный для чтения символ является символом перевода строки, он считывается и отбрасывается. После считывания 29 символов флаг `failbit` устанавливается, если только следующий символ не является символом перевода строки. Таким образом, строка ввода длиной 30 символов или более завершит ввод.

Теперь рассмотрим метод `get(char *, int)`. Сначала он проверяет количество символов, затем признак конца файла и, наконец — является ли очередной символ символом перевода строки. Он не устанавливает флаг `failbit` после считывания максимального количества символов. Тем не менее, можно определить, когда окончание считывания в методе вызвано слишком большим количеством вводимых символов. Можно воспользоваться методом `peek()` (см. следующий раздел) для проверки следующего символа ввода. Если это перевод строки, метод `get()` должен прочитать полную строку. Если это не символ перевода строки, `get()` должен прекратить свое выполнение до достижения конца. Эта методика не всегда срабатывает с методом `getline()`, поскольку он считывает и отбрасывает символы перевода строки, и, следовательно, просмотр следующего символа ничего не даст. Но в случае применения `get()` имеется возможность предпринять определенные действия при считывании неполной строки. Пример такого подхода приведен в следующем разделе. А пока особенности поведения этих методов кратко описаны в табл. 17.6.

Таблица 17.6. Поведение при вводе

Метод	Поведение
`getline(char *, int)`	Устанавливает `failbit`, если ни один символ не прочитан (но перевод строки считается прочитанным символом).
	Устанавливает `failbit`, если прочитано максимальное количество символов, но в строке еще остаются непрочитанные символы
`get(char *, int)`	Устанавливает `failbit`, если ни один символ не прочитан

Другие методы класса `istream`

Другие методы `istream`, помимо уже описанных, включают `read()`, `peek()`, `gcount()` и `putback()`. Функция `read()` считывает заданное количество байтов и сохраняет их в указанном месте. Например, следующие операторы считывают 144 символа из стандартного ввода и помещают их в массив `gross`.

```
char gross[144];
cin.read(gross, 144);
```

В отличие от `getline()` и `get()`, метод `read()` не добавляет нулевой символ к вводу и, следовательно, не преобразует ввод в строковую форму. Метод `read()` не предна-

значен для клавиатурного ввода. Вместо этого чаще всего он применяется в сочетании с функцией `write()` класса `ostream` для файлового ввода и вывода. Возвращаемым типом этого метода является `istream &`, поэтому к его значению можно применять конкатенацию следующим образом:

```
char gross[144];
char score[20];
cin.read(gross, 144).read(score, 20);
```

Функция `peek()` возвращает следующий символ ввода без извлечения его из входного потока. То есть она позволяет взглянуть на следующий символ. Предположим, что требуется выполнить считывание ввода вплоть до первого символа перевода строки или точки, в зависимости от того, что встретится раньше. Функцию `peek()` можно применить для просмотра следующего символа входного потока, чтобы оценить, стоит ли продолжать:

```
char great_input[80];
char ch;
int i = 0;
while ((ch = cin.peek()) != '.' && ch != '\n')
   cin.get(great_input[i++]);
great_input [i] = '\0';
```

Вызов `cin.peek()` выбирает следующий входной символ и присваивает его значение переменной `ch`. Затем условие цикла `while` удостоверяется, что значение `ch` не является ни точкой, ни символом новой строки. Если это так, цикл считывает символ в массив и обновляет его индекс. Когда цикл прерывается, точка или символ новой строки остаются во входном потоке в позиции первого символа, который будет считан следующей операцией ввода. Затем код добавляет нулевой символ к массиву, превращая его в строку.

Метод `gcount()` возвращает количество символов, прочитанных последним методом неформатированного извлечения. Это подразумевает символы, считанные методом `get()`, `getline()`, `ignore()` или `read()`, но не операцией извлечения (>>), которая форматирует ввод в соответствии с определенными типами данных.

Например, предположим, что вы использовали `cin.get(myarray, 80)` для чтения строки в массив `myarray`, и хотите знать, сколько символов было прочитано. Для подсчета символов в массиве можно было бы воспользоваться функцией `strlen()`, но быстрее применить `cin.getcount()`, чтобы выяснить, сколько символов только что было считано из входного потока.

Функция `putback()` вставляет символ обратно в строку ввода. При этом вставленный символ становится первым символом, читаемым следующим оператором ввода. Метод `putback()` принимает один аргумент `char`, представляющий вставляемый символ, и возвращает тип `istream &`, что позволяет осуществлять конкатенацию вызова с другими методами `istream`. Применение `peek()` подобно вызову `get()` для чтения символа с последующим использованием `putback()` для помещения прочитанного символа обратно во входной поток. Однако `putback()` дает возможность вернуть в поток символ, отличающийся от последнего прочитанного.

В листинге 17.14 используются два подхода для чтения и отображения ввода, вплоть до символа # (но, не включая его). Первый подход считывает символы до символа # и затем использует `putback()` для вставки этого символа обратно в поток. Второй подход применяет `peek()` для того, чтобы заглянуть вперед, прежде чем выполнить считывание ввода.

Листинг 17.14. peeker.cpp

```
// peeker.cpp -- некоторые методы istream
#include <iostream>

int main()
{
   using std::cout;
   using std::cin;
   using std::endl;

   // Считывание и отображение символов до символа #
   char ch;

   while(cin.get(ch))                    // завершение по достижении EOF
   {
      if (ch != '#')
         cout << ch;
      else
      {
         cin.putback(ch);                // повторная вставка символа
         break;
      }
   }

   if (!cin.eof())
   {
      cin.get(ch);
      cout << endl << ch << " is next input character.\n";
   }
   else
   {
      cout << "End of file reached.\n"; // достигнут конец файла
      std::exit(0);
   }
   while(cin.peek() != '#')              // "заглядывание" вперед
   {
      cin.get(ch);
      cout << ch;
   }

   if (!cin.eof())
   {
      cin.get(ch);
      cout << endl << ch << " is next input character.\n";
   }
   else
      cout << "End of file reached.\n"; // достигнут конец файла

   return 0;
}
```

Ниже приведен пример выполнения программы из листинга 17.14:

```
I used a #3 pencil when I should have used a #2.
I used a
# is next input character.
3 pencil when I should have used a
# is next input character.
```

Замечания по программе

Давайте подробнее рассмотрим код в листинге 17.14. Первый подход использует цикл `while` для считывания ввода:

```
while(cin.get(ch))        // прерывание по достижении EOF
{
   if (ch != '#')
      cout << ch;
   else
   {
      cin.putback(ch);    // повторная вставка символа
      break;
   }
}
```

Выражение `cin.get(ch)` возвращает `false` при достижении условия конца файла, поэтому эмуляция конца файла с клавиатуры завершает цикл. Если символ # появляется раньше, то программа помещает его обратно во входной поток и с помощью оператора `break` завершает цикл.

Второй подход проще:

```
while(cin.peek() != '#')  // "заглядывание" вперед
{
   cin.get(ch);
   cout << ch;
}
```

Программа просматривает следующий символ. Если он не является символом #, программа считывает следующий символ, отображает его и просматривает следующий символ. Это продолжается до тех пор, пока не появится завершающий символ.

А теперь взглянем, как и было обещано, на пример, использующий `peek()` для определения того, была ли считана вся строка целиком (см. листинг 17.15). Если во входном массиве умещается только часть строки, программа отбрасывает остальную ее часть.

Листинг 17.15. `truncate.cpp`

```
// truncate.cpp — использование get() для усечения входной строки в случае
необходимости
#include <iostream>
const int SLEN = 10;
inline void eatline() { while (std::cin.get() != '\n') continue; }
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;

   char name[SLEN];
   char title[SLEN];
   cout << "Enter your name: ";     // приглашение для ввода имени
   cin.get(name,SLEN);
   if (cin.peek() != '\n')
      cout << "Sorry, we only have enough room for "
           << name << endl;         // вывод сообщения о недостатке места

   eatline();
   cout << "Dear " << name << ", enter your title: \n"; //приглашение для ввода должности
   cin.get(title,SLEN);
```

```
    if (cin.peek() != '\n')
        cout << "We were forced to truncate your title.\n";
              // вынужденное усечение названия должности
    eatline();
    cout << " Name: " << name
         << "\nTitle: " << title << endl;
    return 0;
}
```

Ниже приведен пример выполнения программы из листинга 17.15:

```
Enter your name: Ella Fishsniffer
Sorry, we only have enough room for Ella Fish
Dear Ella Fish, enter your title:
Executive Adjunct
We were forced to truncate your title.
 Name: Ella Fish
Title: Executive
```

Обратите внимание, что следующий код имеет разный смысл в зависимости от того, считывает ли первый оператор ввода полную строку:

```
while (cin.get() != '\n') continue;
```

Если `get()` считывает полную строку, он все же оставляет на месте символ перевода строки, и этот код читает и отбрасывает символ новой строки. Если же `get()` считывает только часть строки, этот код читает и отбрасывает остаток строки. Если бы остаток строки не был отброшен, следующий оператор ввода начал бы считывание с начала оставшейся части первой строки ввода. В данном примере это привело бы к чтению строки `sniffer` в массив `title`.

Файловый ввод и вывод

Большинство компьютерных программ работает с файлами. Текстовые процессоры создают файлы документов. Программы баз данных создают файлы и ищут в них информацию. Компиляторы считывают файлы исходного кода и генерируют исполняемые файлы. Сам по себе файл — это группа байтов, сохраненных на некотором устройстве, возможно, магнитной ленте, оптическом диске или жестком диске. Как правило, операционная система управляет файлами, отслеживая их местоположение, размеры, дату их создания и т.п. Если только программирование не выполняется на уровне операционной системы, обычно не нужно беспокоиться о подобных нюансах. Все что требуется — это способ подключения программы к файлу, способ считывания программой его содержимого и способ создания и записи файлов программой.

рую поддержку файлов, но значительно более ограниченную, чем явный ввод-вывод, осуществляемый из программы. К тому же перенаправление обеспечивается операционной системой, а не C++, поэтому оно доступно не во всех системах. В этой книге уже затрагивалась тема файлового ввода-вывода, а в этой главе она освещается более подробно.

Пакет классов ввода-вывода C++ управляет файловым вводом и выводом в основном так же, как он делает это со стандартным вводом и выводом. Чтобы записывать в файл, вы создаете объект `ofstream` и используете такие его методы, как операция вставки << или `write()`. Чтобы читать из файла, вы создаете объект `ifstream` и применяете методы `istream` наподобие операции извлечения >> и `get()`. Однако файлы

требуют больше внимания, нежели стандартный ввод и вывод. Например, только что открытый файл нужно ассоциировать с потоком. Файл можно открыть в режиме только для чтения, только для записи либо для чтения и записи. Если вы записываете в файл, может потребоваться создание нового файла, замена старого либо добавление информации в существующий файл. Или же может возникнуть необходимость в перемещении по файлу назад и вперед. Чтобы помочь в решении этих задач, в C++ определено несколько новых классов в заголовочном файле `fstream` (бывший `fstream.h`), включая класс `ifstream` для файлового ввода и класс `ofstream` для файлового вывода. В C++ также определен класс `fstream` для одновременного файлового ввода и вывода. Эти классы являются производными от классов, определенных в заголовочном файле `iostream`, поэтому объекты новых классов могут использовать методы, которые уже были изучены ранее.

Простой файловый ввод-вывод

Предположим, программа должна выполнять запись в файл. Понадобится предпринять следующие действия.

1. Создать объект `ofstream` для управления выходным потоком.
2. Ассоциировать этот объект с конкретным файлом.
3. Использовать объект так же, как нужно было бы использовать `cout`. Единственным отличием будет то, что вывод направляется в файл вместо экрана.

Чтобы достичь этого, нужно начать с подключения заголовочного файла `fstream`. Его подключение в большинстве, хотя и не во всех реализациях, автоматически подключает файл `iostream`, поэтому явное подключение `iostream` не обязательно. Затем нужно объявить объект типа `ofstream`:

```
ofstream fout;              // создание объекта fout типа ofstream
```

Именем объекта может быть любое допустимое в C++ имя, такое как `fout`, `outFile`, `cgate` или `didi`.

Затем этот объект нужно ассоциировать с конкретным файлом. Это можно сделать с помощью метода `open()`. Предположим, например, что требуется открыть файл `jar.txt` для вывода. Это можно было бы сделать следующим образом:

```
fout.open("jar.txt");      // связывание fout с файлом jar.txt
```

Эти два шага (создание объекта и ассоциация файла с ним) можно совместить в одном операторе, используя другой конструктор:

```
ofstream fout("jar.txt"); // создание объекта fout и его ассоциирование
                          // с файлом jar.txt
```

После того, как все это сделано, `fout` (или любое другое выбранное вами имя) можно будет использовать таким же образом, как и `cout`. Например, если требуется поместить слова `Dull Data` в этот файл, это можно сделать следующим образом:

```
fout << "Dull Data";
```

Действительно, поскольку `ostream` — это базовый класс для `ofstream`, можно применять все методы `ostream`, включая разнообразные операции вставки, а также методы форматирования и манипуляторы. Класс `ofstream` использует буферизованный вывод, поэтому при создании объекта типа `ofstream`, такого как `fout`, программа выделяет пространство для выходного буфера. Если вы создадите два объекта `ofstream`, программа создаст два буфера — по одному для каждого объекта. Объект `ofstream`,

подобный fout, накапливает выходные данные программы байт за байтом, а затем, когда буфер заполняется, передает его содержимое в файл назначения. Поскольку дисководы спроектированы для передачи данных более крупными порциями, а не побайтно, буферизованный подход значительно увеличивает скорость передачи данных из программы в файл.

Такое открытие файла для вывода создает новый файл, если файла с указанным именем не существует. Если же файл с этим именем существовал ранее, то действие по его открытию усекает его так, чтобы вывод начинался в пустой файл. Позднее в этой главе будет показано, как открыть существующий файл и сохранить его содержимое.

Внимание!

Открытие файла для вывода в режиме по умолчанию автоматически усекает его до нулевого размера, по существу уничтожая его предыдущее содержимое.

Действия для выполнения чтения из файла во многом подобны тем, которые необходимы для выполнения записи в файл.

1. Создать объект ifstream для управления входным потоком.
2. Ассоциировать этот объект с конкретным файлом.
3. Использовать объект так же, как нужно было бы использовать cin.

Шаги для чтения из файла похожи на те, которые нужно выполнить для записи в файл. Для начала, конечно, нужно подключить заголовочный файл fstream. Затем необходимо объявить объект ifstream и ассоциировать его с именем файла. Для этого можно использовать два оператора или же один:

```
// Два оператора
ifstream fin;                  // создать объекта fin типа ifstream
fin.open("jellyjar.txt");      // открытие файла jellyjar.txt для чтения

// Один оператор
ifstream fis("jamjar.dat");    // создание объекта fis и его ассоциирование
                               // с файлом jamjar.txt
```

Затем объект fin или fis можно использовать почти так же, как cin. Например, можно использовать следующий код:

```
char ch;
fin >> ch;                     // считывание символа из файла jellyjar.txt
char buf[80];
fin >> buf;                    // считывание слова из файла
fin.getline(buf, 80);          // считывание строки из файла
string line;
getline(fin, line);            // считывание из файла в строковый объект
```

Ввод, как и вывод, также буферизуется, поэтому создание объекта ofstream, такого как fin, создает входной буфер, которым управляет объект fin. Как и в случае вывода, буферизация обеспечивает гораздо более быстрое перемещение данных, чем при побайтной передаче.

Соединение с файлом закрывается автоматически, когда объекты ввода и вывода уничтожаются, например, по завершении программы. Кроме того, соединение с файлом можно закрыть явно, используя для этого метод close():

```
fout.close();                  // закрытие соединения вывода с файлом
fin.close();                   // закрытие соединения ввода с файлом
```

Закрытие такого соединения не уничтожает поток; оно просто отключает его от файла. Однако средства управления потоком никуда не деваются. Например, объект `fin` продолжает существовать, как и входной буфер, которым он управляет. Как будет показано позже, этот поток можно заново подключить к тому же или к другому файлу.

Рассмотрим краткий пример. Программа в листинге 17.16 запрашивает имя файла. Она создает файл с этим именем, записывает в него некоторую информацию и закрывает файл. Закрытие файла очищает буфер, тем самым гарантируя обновление файла. Затем программа открывает тот же файл для чтения и отображает его содержимое. Обратите внимание, что программа использует `fin` и `fout` таким же образом, как если бы применялись `cin` и `cout`. Кроме того, программа считывает имя файла в объект `string`, а затем использует метод `c_str()` для передачи конструкторам `ofstream` и `ifstream` аргументов — строк в стиле C.

Листинг 17.16. `fileio.cpp`

```
#include <iostream>  // для многих систем не требуется
#include <fstream>
#include <string>

int main()
{
   using namespace std;
   string filename;
   cout << "Enter name for new file: ";       // запрос имени нового файла
   cin >> filename;

   // Создание объекта выходного потока для нового файла и назначение ему имени fout
   ofstream fout(filename.c_str());
   fout << "For your eyes only!\n";           // запись в файл
   cout << "Enter your secret number: ";      // вывод на экран
   float secret;
   cin >> secret;
   fout << "Your secret number is " << secret << endl;
   fout.close();                              // закрытие файла

   // Создание объекта входного потока для нового файла и назначение ему имени fin
   ifstream fin(filename.c_str());
   cout << "Here are the contents of " << filename << ":\n";
   char ch;
   while (fin.get(ch))                        // чтение символа из файла
      cout << ch;                             // и его вывод на экран
   cout << "Done\n";
   fin.close();

   return 0;
}
```

Ниже приведен пример выполнения программы из листинга 17.16:

```
Enter name for new file: pythag
Enter your secret number: 3.14159
Here are the contents of pythag:
For your eyes only!
Your secret number is 3.14159
Done
```

Если вы просмотрите каталог, содержащий программу, то найдете там файл `pythag` и, загрузив его в любом текстовом редакторе, увидите то же содержимое, что и выводе программы.

Проверка потока и is_open()

Классы файловых потоков C++ наследуют член, описывающий состояние потока, от класса ios_base. Этот член, как упоминалось ранее, хранит информацию, отражающую состояние потока — о том, что все в порядке, что достигнут конец файла, о том, произошел ли сбой операции ввода-вывода, и т.д. Если все в порядке, состояние потока равно нулю (отсутствие новостей — уже хорошая новость). Разнообразные другие состояния описываются установкой конкретных битов в единицу. Классы файловых потоков также наследуют методы ios_base, которые сообщают о состоянии потока и перечислены в табл. 17.4. Можно проверить состояние потока, чтобы выяснить, успешно ли завершилась последняя операция с этим потоком. Для файловых потоков это включает в себя проверку успешности или неудачи операции открытия файла. Например, попытка открытия для ввода несуществующего файла устанавливает флаг failbit. Поэтому можно было бы выполнить проверку следующим образом:

```
fin.open(argv[file]);
if (fin.fail())      // попытка открытия не удалась
{
   ...
}
```

Или же, поскольку объект ifstream, подобно istream, преобразуется в тип bool, когда ожидается именно этот тип, можно было бы использовать следующий код:

```
fin.open(argv[file]);
if (!fin)            // попытка открытия не удалась
{
   ...
}
```

Однако новые реализации C++ предлагают лучший способ проверки того, открыт ли файл — метод is_open().

```
if (!fin.is_open())  // попытка открытия не удалась
{
   ...
}
```

Преимущество этого способа состоит в том, что он проверяет также наличие некоторых незначительных проблем, которые остаются незамеченными другими формами проверки, как указано в следующей врезке "Внимание!".

Внимание!

В прошлом проверка успешности открытия файла выполнялась следующим образом:

```
if(fin.fail()) ...        // неудача открытия
if(!fin.good()) ...       // неудача открытия
if (!fin) ...             // неудача открытия
```

Объект fin, когда он используется в условии if, преобразуется в false, если fin.good() возвращает false, и в true — в остальных случаях, поэтому две приведенные формы эквивалентны. Однако эти тесты не могут правильно распознать ситуацию, когда предпринимается попытка открытия файла с неподходящим режимом файла (см. раздел "Режимы файла" далее в настоящей главе.) Метод is_open() перехватывает ошибки подобного рода, наряду с теми, которые перехватываются методом good(). Однако в старых реализациях C++ метод is_open() отсутствует.

Открытие нескольких файлов

Иногда может требоваться, чтобы программа открывала более одного файла. Стратегия открытия нескольких файлов зависит от того, как они будут использоваться. Если требуется, чтобы два файла были открыты одновременно, нужно создать отдельный поток для каждого файла. Например, программа, которая сравнивает два отсортированных файла и отправляет результат в третий, должна создать два объекта `ifstream` для двух входных файлов и один объект `ofstream` — для выходного файла. Количество файлов, которые можно открыть одновременно, зависит от операционной системы.

Однако можно запланировать последовательную обработку файлов. Предположим, что требуется подсчитать, сколько раз имя появляется в наборе из 10 файлов. В этом случае можно открыть единственный поток и по очереди ассоциировать его с каждым из этих файлов. При этом ресурсы компьютера используются экономнее, чем при открытии отдельного потока для каждого файла. Чтобы применить такой подход, нужно объявить объект `ifstream` без его инициализации, а затем с помощью метода `open()` ассоциировать поток с файлом. Например, последовательное считывание двух файлов можно было бы организовать следующим образом:

```
ifstream fin;              // создание потока конструктором по умолчанию
fin.open("fat.txt");       // ассоциирование потока с файлом fat.txt
...                        // выполнение каких-либо действий
fin.close();               // разрыв связи потока с файлом fat.txt
fin.clear();               // сброс fin (может не требоваться)
fin.open("rat.txt");       // ассоциирование потока с файлом rat.txt
...
fin.close();
```

Вскоре мы рассмотрим пример, но сначала изучим технологию передачи списка файлов программе способом, позволяющим программе применять цикл для их обработки.

Обработка командной строки

Программы, обрабатывающие файлы, часто используют аргументы командной строки для идентификации файлов. *Аргументы командной строки* — это параметры, вводимые в командной строке после команды. Например, чтобы подсчитать количество слов в некоторых файлах в системе Unix или Linux, в приглашении командной строки понадобится ввести следующую команду:

```
wc report1 report2 report3
```

Здесь `wc` — имя программы, а `report1`, `report2` и `report3` — имена файлов, переданные программе в качестве аргументов командной строки.

В C++ имеется механизм, который позволяет программам, запущенным из среды командной строки, получать доступ к аргументам командной строки. Можно использовать следующий альтернативный заголовок функции `main()`:

```
int main(int argc, char *argv[])
```

Аргумент `argc` представляет количество аргументов в командной строке. Счетчик включает имя самой команды. Переменная `argv` — это указатель на указатель на `char`. Это звучит несколько абстрактно, но `argv` можно трактовать как массив указателей на аргументы командной строки, причем `argv[0]` указывает на первый символ строки, содержащей имя самой команды, `argv[1]` — указатель на первый символ строки, содержащей первый аргумент командной строки, и т.д. То есть `argv[0]` — первая строка команды и т.д.

Например, предположим, что имеется следующая командная строка:

```
wc report1 report2 report3
```

В этом случае `argc` будет равно `4`, `argv[0]` — `wc`, `argv[1]` — `report1` и т.д. Следующий цикл будет выводить каждый аргумент командной строки в отдельной строке экрана:

```
for (int i = 1; i < argc; i++)
  cout << argv[i] << endl;
```

Если начать с `i = 1`, то будут выведены только аргументы командной строки, а если начать с `i = 0`, будет выведено и имя команды.

Конечно, аргументы командной строки тесно взаимосвязаны с операционными системами, ориентированными на командную строку, такими как режим командной строки Windows, Unix и Linux. Другие среды также могут допускать использование аргументов командной строки.

- Многие интегрированные среды разработки (Integrated Development Environments — IDE) для Windows имеют опцию, позволяющую передавать им аргументы командной строки. Обычно требуется осуществлять навигацию по серии пунктов меню, чтобы добраться до поля, в котором можно ввести аргументы командной строки. Конкретная последовательность шагов варьируется в зависимости от поставщика и от версии, поэтому за подробными инструкциями следует обращаться к документации.
- Многие IDE-среды в Windows могут генерировать исполняемые файлы, которые запускаются в режиме командной строки Windows.

Код в листинге 17.17 сочетает технологию командной строки с технологиями файловых потоков для подсчета количества символов в файлах, перечисленных в командной строке.

Листинг 17.17. `count.cpp`

```
// count.cpp -- подсчет символов в списке файлов
#include <iostream>
#include <fstream>
#include <cstdlib>                  // для exit()
int main(int argc, char * argv[])
{
   using namespace std;
   if (argc == 1)                   // выход при отсутствии аргументов
   {
      cerr << "Usage: " << argv[0] << " filename[s]\n";
      exit(EXIT_FAILURE);
   }
   ifstream fin;                    // открытие потока
   long count;
   long total = 0;
   char ch;
   for (int file = 1; file < argc; file++)
   {
      fin.open(argv[file]);         // подключение потока к argv[file]
      if (!fin.is_open())
      {
         cerr << "Could not open " << argv[file] << endl;  // не удается открыть файл
         fin.clear();
         continue;
      }
```

```
        count = 0;
        while (fin.get(ch))
            count++;
        cout << count << " characters in " << argv[file] << endl;
                        // количество символов в файле
        total += count;
        fin.clear();          // требуется для некоторых реализаций
        fin.close();          // отключение от файла
    }
    cout << total << " characters in all files\n"; // количество символов во всех файлах
    return 0;
}
```

Некоторые реализации C++ требуют вызова `fin.clear()` в конце программы, а другие — нет. Это зависит от того, сбрасывается ли состояние потока автоматически при ассоциировании нового файла с объектом типа `ifstream`. Использование `fin.clear()` не повредит, даже если в этом нет необходимости.

Например, в системе Linux код из листинга 17.17 можно было бы скомпилировать в исполняемый файл `count.exe`. Пример его выполнения мог бы выглядеть следующим образом:

```
$ a.out
Usage: a.out filename[s]
$ a.out paris rome
3580 characters in paris
4886 characters in rome
8466 characters in all files
$
```

Обратите внимание, что программа использует `cerr` для вывода сообщений об ошибках. Небольшой нюанс заключается в том, что сообщение использует `argv[0]` вместо `count.exe`:

```
cerr << "Usage: " << argv[0] << " filename[s]\n";
```

Это дает возможность программе в случае изменения имени исполняемого файла автоматически использовать новое имя.

Программа применяет метод `is_open()` для проверки того, что запрошенный файл удалось открыть. Рассмотрим этот момент подробнее.

Режимы файла

Режим файла описывает, как файл будет использоваться: для чтения, записи, добавления информации и т.п. При ассоциировании потока с файлом либо инициализацией объекта файлового потока именем файла, либо с помощью метода `open()`, можно указать второй аргумент, описывающий режим файла:

```
ifstream fin("banjo", mode1);       // конструктор с аргументом режима
ofstream fout();
fout.open("harp", mode2);           // open() с аргументом режима
```

Класс `ios_base` определяет тип `openmode`, представляющий режим. Подобно типам `fmtflags` и `iostate`, он представляет собой тип `bitmask`. (В старые времена он имел тип `int`.) Для указания режима можно выбрать константу из числа определенных в классе `ios_base`.

Константы и их назначения перечислены в табл. 17.7. Файловый ввод-вывод C++ претерпел некоторые изменения для обеспечения его совместимости с файловым вводом-выводом ANSI C.

Таблица 17.7. Константы режима файла

Константа	Описание
`ios_base::in`	Открыть файл для чтения
`ios_base::out`	Открыть файл для записи
`ios_base::ate`	Перейти к концу файла после его открытия
`ios_base::app`	Добавлять в конец файла
`ios_base::trunc`	Усечь файл, если он существует
`ios_base::binary`	Двоичный файл

Если конструкторы `ifstream` и `ofstream`, а также методы `open()` принимают два аргумента, каким образом следует трактовать их вызов с одним аргументом в предшествующих примерах? Как можно предположить, прототипы этих функций-членов класса предусматривают значения по умолчанию для второго аргумента (аргумента, описывающего режим файла).

Например, метод `open()` и конструктор `ifstream` в качестве значения по умолчанию для аргумента режима используют `ios_base::in` (открыть для чтения), а метод `open()` и конструктор `ofstream` в качестве значения по умолчанию применяют `ios_base::out | ios_base::trunc` (открыть для чтения и усечь файл). Битовая операция "ИЛИ" (`|`) служит для объединения двух битовых значений в одно, которое может быть использовано для одновременной установки обоих битов. В классе `fstream` режим по умолчанию не предусмотрен, поэтому при создании объектов данного класса режим нужно указывать явно.

Обратите внимание, что флаг `ios_base::trunc` означает, что для приема вывода из программы существующий файл усекается, т.е. его предыдущее содержимое отбрасывается. Хотя такое поведение снижает риск переполнения дискового пространства, легко представить себе ситуации, в которых стирание содержимого файла при его открытии нежелательно. Конечно же, в C++ предусмотрены и другие варианты. Если, например, требуется сохранить содержимое файла и добавить новую информацию в его конец, можно воспользоваться режимом `ios_base::app`:

```
ofstream fout("bagels", ios_base::out | ios_base::app);
```

И снова этот код использует операцию `|` для объединения режимов. Поэтому `ios_base::out | ios_base::app` означает, что нужно включить и режим `out`, и режим `app` (рис. 17.6).

Более старые реализации C++ могут иметь некоторые различия. Например, некоторые из них позволяют пропустить `ios_base::out` в предыдущем примере, а некоторые — нет. Если не используется режим по умолчанию, более безопасный подход — явное указание всех элементов режима. Некоторые компиляторы поддерживают не все варианты, перечисленные в табл. 17.8, а некоторые могут предлагать дополнительные, помимо перечисленных. Одно из следствий этих различий заключается в том, что, возможно, придется внести некоторые изменения в последующие примеры, чтобы использовать их в конкретной системе. Отрадно то, что непрекращающаяся разработка стандарта C++ обеспечивает все большее единообразие.

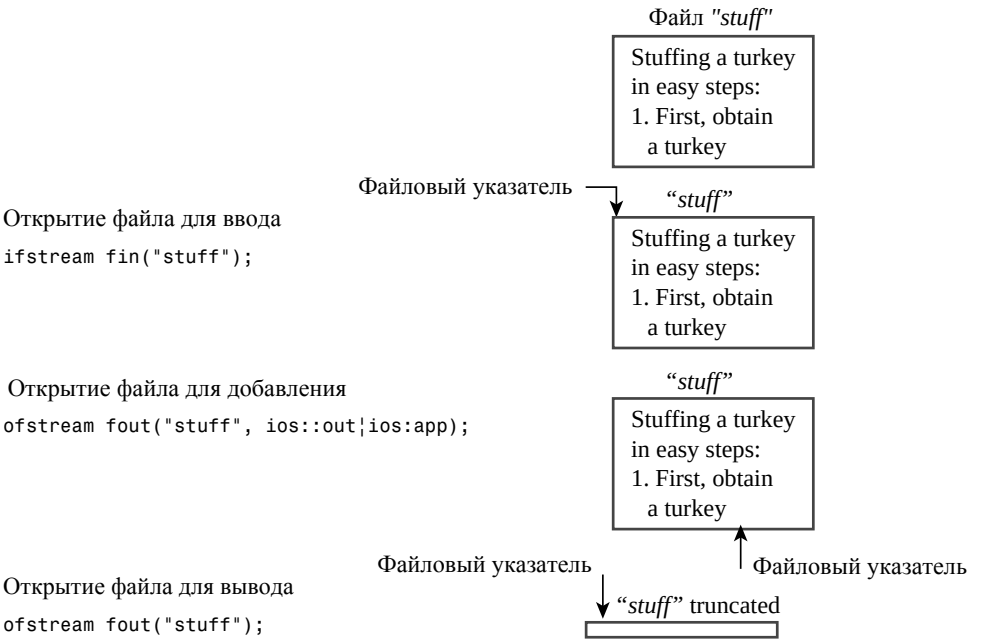


***Рис. 17.6.** Некоторые режимы открытия файла*

Стандарт C++ определяет части файлового ввода-вывода в терминах их эквивалентов из стандарта ввода-вывода ANSI C. Оператор C++ вроде

```
ifstream fin(filename, c++mode);
```

реализуется, как если бы он использовал функцию `fopen()` из C:

```
fopen(filename, cmode);
```

Здесь *c++mode* — значение типа `openmode`, такое как `ios_base::in`, а *cmode* — соответствующая строка режима C, подобная `"r"`. Соответствия между режимами C++ и C показаны в табл. 17.8. Обратите внимание, что аргумент `ios_base::out` сам по себе вызывает усечение, но не делает этого, если применяется в комбинации с `ios_base::in`. Не перечисленные комбинации, такие как `ios_base::in | ios_base::trunc`, препятствуют открытию файла. Метод `is_open()` выявляет этот сбой.

Таблица 17.8. Режимы открытия файлов C++ и C

Режим C++	Режим C	Описание
`ios base::in`	`"r"`	Открыть для чтения
`ios_base::out`	`"w"`	То же, что и `ios_base::out | ios_base::trunc`
`ios_base::out ¦ ios_base::trunc`	`"w"`	Открыть для записи с усечением существующего файла
`ios_base::out | ios_base::app`	`"a"`	Открыть для записи с разрешением только на добавление

Окончание табл. 17.8

Режим C++	Режим C	Описание
`ios_base::in | ios_base::out`	`"r+"`	Открыть для чтения и записи с разрешением на запись в произвольном месте файла
`ios_base::in | ios_base::out | ios_base::trunc`	`"w+"`	Открыть для чтения и записи с усечением существующего файла
c++mode `| ios_base::binary`	`"`*cmode*`b"`	Открыть в режиме *c++mode* или соответствующем режиме *cmode* и в бинарном (не текстовом) режиме. Например, `ios_base::in | ios_base::binary` становится `"rb"`
c++mode `| ios_base::ate`	`"`*cmode*`"`	Открыть в указанном режиме и перейти к концу файла. В C вместо кода режима используется отдельный вызов функции. Например, `ios_base::in | ios_base::ate` преобразуется в режим `"r"` с последующим вызовом функции `fseek(file, 0, SEEK_END)`

Обратите внимание, что и `ios_base::ate`, и `ios_base::app` помещает файловый указатель в конец открытого файла. Разница между этими двумя режимами состоит в том, что `ios_base::app` позволяет только добавлять данные в конец файла, в то время как `ios_base::ate` просто устанавливает указатель на конец файла.

Ясно, что существует множество возможных комбинаций режимов. Мы рассмотрим несколько наиболее типичных.

Добавление к файлу

Рассмотрим программу, которая дописывает данные в конец файла. Программа поддерживает файл, содержащий список гостей. Когда она начинает выполнение, то отображает текущее содержимое файла, если он уже существует. Она может использовать метод `is_open()` после попытки открытия файла для проверки, существует ли он. Затем программа открывает файл для вывода, используя режим `ios_base::app`. Затем она принимает ввод с клавиатуры, чтобы дописать информацию в файл. И, наконец, программа отображает измененное содержимое файла. Код в листинге 17.18 иллюстрирует, как все это можно реализовать на практике. Обратите внимание, как программа использует метод `is_open()` для проверки успешности открытия файла.

> **На заметку!**
>
> Файловый ввод-вывод — возможно, наименее стандартизованный аспект C++ в его ранних реализациях, и даже сейчас многие компиляторы не полностью соответствуют современному стандарту. Например, некоторые используют такой режим, как `nocreate`, который не является частью современного стандарта. Вдобавок лишь некоторые компиляторы требуют вызова `fin.clear()` перед открытием того же файла для чтения во второй раз.

Листинг 17.18. `append.cpp`

```
// append.cpp -- добавление информации в файл
#include <iostream>
#include <fstream>
#include <string>
#include <cstdlib>  // для exit()
```

```
const char * file = "guests.txt";
int main()
{
   using namespace std;
   char ch;

   // Отображение начального содержимого
   ifstream fin;
   fin.open(file);

   if (fin.is_open())
   {
      cout << "Here are the current contents of the "
         << file << " file:\n";
      while (fin.get(ch))
         cout << ch;
      fin.close();
   }

   // Добавление новых имен
   ofstream fout(file, ios::out | ios::app);
   if (!fout.is_open())
   {
     cerr << "Can't open " << file << " file for output.\n";  // не удается открыть файл
     exit(EXIT_FAILURE);
   }
   cout << "Enter guest names (enter a blank line to quit):\n";
   string name;
   while (getline(cin,name) && name.size() > 0)
   {
      fout << name << endl;
   }
   fout.close();

   // Отображение измененного файла
   fin.clear();  // не обязательно для некоторых компиляторов
   fin.open(file);
   if (fin.is_open())
   {
      cout << "Here are the new contents of the "
           << file << " file:\n";
      while (fin.get(ch))
         cout << ch;
      fin.close();
   }
   cout << "Done.\n";
   return 0;
}
```

Вот как выглядит пример первого запуска программы из листинга 17.18:

```
Enter guest names (enter a blank line to quit):
Genghis Kant
Hank Attila
Charles Bigg

Here are the new contents of the guests.txt file:
Genghis Kant
Hank Attila
Charles Bigg
Done.
```

При первом запуске файл `guests.txt` не существовал, поэтому программа не отобразила его исходное содержимое.

Однако при следующем запуске программы файл `guests.txt` уже будет существовать, поэтому программа сначала покажет его содержимое. Кроме того, обратите внимание, что новые данные добавляются в конец файла, а не заменяют старые:

```
Here are the current contents of the guests.txt file:
Genghis Kant
Hank Attila
Charles Bigg
Enter guest names (enter a blank line to quit):
Greta Greppo
LaDonna Mobile
Fannie Mae

Here are the new contents of the guests.txt file:
Ghengis Kant
Hank Attila
Charles Bigg
Greta Greppo
LaDonna Mobile
Fannie Mae
Done.
```

Содержимое файла `guests.txt` можно просмотреть в любом текстовом редакторе, включая тот, который применяется для написания исходного кода.

Бинарные файлы

Сохранять данные в файле можно в текстовом или в бинарном формате. Текстовая форма означает хранение всех данных — даже чисел — в виде текста. Например, сохранение значения `-2.324216e+07` в текстовой форме означает сохранение 13 символов, используемых для записи этого числа. Это требует преобразования внутреннего компьютерного представления числа с плавающей точкой в символьную форму, что и делает операция вставки <<. С другой стороны, бинарный формат означает хранение внутреннего компьютерного представления значения. То есть вместо хранения символов компьютер сохраняет (как правило) 64-разрядное представление значения типа `double`. Для символа бинарное представление совпадает с текстовым представлением — бинарным представлением ASCII-кода (или его эквивалента) символа. Однако для чисел бинарное представление значительно отличается от символьного (рис. 17.7).

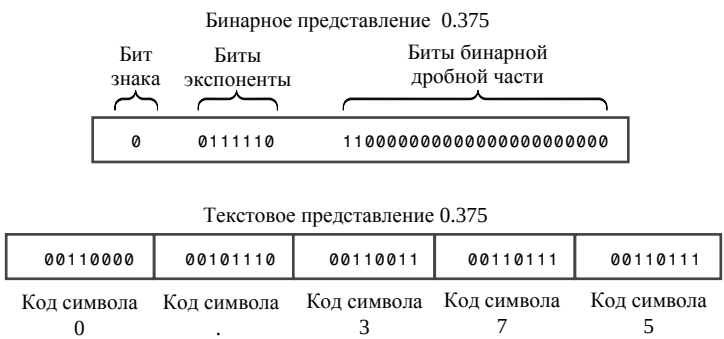


Рис. 17.7. *Бинарное и текстовое представление числа с плавающей точкой*

Каждый формат обладает своими преимуществами. Текстовый формат легко читать. С ним можно использовать обычный редактор или текстовый процессор для чтения и редактирования текстового файла. Текстовый файл несложно передать из одной системы в другую. Бинарный формат более точен для представления чисел, поскольку сохраняет точное внутреннее представление значения. Его применение позволяет избежать ошибок, связанных с преобразованием или округлением. Сохранение данных в бинарном формате может выполняться быстрее, поскольку никаких преобразований не происходит и данные можно сохранять более крупными порциями. К тому же, в зависимости от природы данных, обычно бинарный формат требует меньше места. Однако их передача в другую систему может оказаться проблемой, если в этой системе используется другое внутреннее представление значений. Даже разные компиляторы в одной и той же системе могут применять различное внутреннее представление структурных макетов. В этих случаях вам (или кому-то другому) может потребоваться написать программу для преобразования одного формата данных в другой.

Рассмотрим конкретный пример. Предположим, что есть следующее определение и объявление структуры:

```
const int LIM = 20;
struct planet
{
   char name[LIM];       // название планеты
   double population;    // население
   double g;             // ускорение свободного падения
};
planet pl;
```

Чтобы сохранить содержимое структуры `pl` в текстовом формате, можно использовать следующий код:

```
ofstream fout("planets.dat", ios_base:: out | ios_base::app);
fout << pl.name << " " << pl.population << " " << pl.g << "\n";
```

Обратите внимание, что вам придется указывать каждый член структуры явно, применяя операцию членства, и для обеспечения читабельности разделять соседние данные. Если структура содержит, скажем, 30 членов, это может оказаться утомительным.

Чтобы сохранить ту же информацию в бинарной форме, можно использовать следующий код:

```
ofstream fout("planets.dat",
              ios_base:: out | ios_base::app | ios_base::binary);
fout.write( (char *) &pl, sizeof pl);
```

Этот код сохраняет всю структуру как единое целое, используя внутреннее компьютерное представление данных. Этот файл не удастся прочитать как текстовый, но информация будет сохранена в более компактном и точном виде, чем в текстовой форме. К тому же приведенный код определенно легче ввести. Этот подход отличается следующими аспектами.

- Он использует бинарный режим файла.
- Он использует функцию-член `write()`.

Рассмотрим эти изменения подробнее.

В ряде систем, таких как Windows, поддерживаются два формата файлов: текстовый и бинарный. Если требуется сохранить данные в бинарной форме, нужно ис-

пользовать бинарный формат файлов. В языке C++ это достигается за счет указания константы `ios_base::binary` в режиме файла. Если вы хотите знать, почему это необходимо в системе Windows, прочтите следующую врезку "Бинарные и текстовые файлы".

Бинарные и текстовые файлы

Использование бинарного режима файла вынуждает программу передавать данные из памяти в файл и обратно без какого-либо скрытого преобразования. Это может не подойти для текстового режима, используемого по умолчанию. Например, рассмотрим текстовые файлы Windows. Они представляют новую строку комбинацией двух символов: возврат каретки и перевод строки. Текстовые файлы Macintosh представляют новую строку только с помощью символа возврата каретки. Файлы Unix и Linux представляют новую строку только символом перевода строки. В языке C++, который вырос на почве Unix, новая строка также представляется символом перевода строки. Программы на C++ в среде Windows, когда записывают файл в текстовом режиме, автоматически преобразуют новую строку C++ в комбинацию "возврат каретки, перевод строки", а программы на C++ в среде Macintosh преобразуют новую строку в символ возврата каретки. При чтении текстового файла эти программы выполняют обратное преобразование локальной новой строки в форму, принятую в C++. Текстовый формат файлов может вызвать проблемы с бинарными данными, потому что байт в середине значения типа `double` может иметь тот же битовый шаблон, что и ASCII-код символа перевода строки. К тому же существуют различия в способе обнаружения конца файла. Поэтому при сохранении данных в бинарном формате нужно применять бинарный режим. (Системы Unix поддерживают только один режим файлов, поэтому в них бинарный режим совпадает с текстовым.)

Чтобы сохранить данные в бинарной форме вместо текстовой, можно воспользоваться функцией-членом `write()`. Вспомните, что этот метод копирует указанное количество байт из памяти в файл. Ранее в этой главе он применялся для копирования текста, но он будет побайтно копировать данные любого типа без каких-либо преобразований. Например, если передать ему адрес переменной типа `long` и указать, что необходимо скопировать 4 байта, он буквально скопирует в файл 4 байта, составляющие значение типа `long`, не преобразуя его в текст. Единственным неудобством будет то, что придется использовать приведение адреса к типу указателя на значение типа `char`. Тот же подход можно использовать для копирования всей структуры `planet`. Чтобы получить количество байт, которые должны быть записаны, следует применить операцию `sizeof`:

```
fout.write((char *)&pl, sizeof pl);
```

Этот оператор вынуждает программу обратиться к адресу структуры `pl` и скопировать 36 байт (значение выражения `sizeof pl`), начиная с указанного адреса, в файл, подключенный к `fout`.

Чтобы восстановить информацию из файла, нужно использовать соответствующий метод `read()` с объектом `ifstream`:

```
ifstream fin("planets.dat", ios_base::in | ios_base::binary);
fin.read((char *) &pl, sizeof pl);
```

Этот код копирует `sizeof pl` байт из файла в структуру `pl`. Тот же подход можно использовать с классами, у которых нет виртуальных функций. В этом случае будут сохранены только данные-члены, но не методы. Если же класс имеет виртуальные методы, то скрытый указатель на таблицу указателей виртуальных функций также будет скопирован. Поскольку при следующем запуске программы таблица виртуальных

функций может быть размещена в другом месте, то копирование старого значения указателя из файла в объекты может привести к хаосу. (Прочитайте также врезку "На заметку!" после упражнения 6 по программированию.)

Совет

Функции-члены `read()` и `write()` дополняют друг друга. Функция `read()` используется для восстановления данных, которые были записаны в файл методом `write()`.

Код в листинге 17.19 применяет эти методы для создания и чтения бинарного файла. По форме эта программа подобна приведенной в листинге 17.18, но использует `write()` и `read()` вместо операции вставки и метода `get()`. Она также применяет манипуляторы для форматирования экранного вывода.

На заметку!

Хотя концепция бинарного файла является частью ANSI C, в некоторых реализациях C и C++ не предусмотрена поддержка бинарного режима файлов. Причина этого упущения в том, что в ряде систем существует только один тип файлов, поэтому можно использовать бинарные операции вроде `read()` и `write()` с файлами стандартного формата. Поэтому, если ваша реализация не воспринимает `ios_base::binary` как допустимую константу, ее можно просто исключить из программы. Если ваша реализация не поддерживает манипуляторы `fixed` и `right`, можно применить `cout.setf(ios_base::fixed, ios_base::floatfield)` и `cout.setf(ios_base::right, ios_base::adjustfield)`. Кроме того, возможно, придется заменить `ios_base` вариантом `ios`. Другие компиляторы, особенно более старые, могут иметь иные отличия.

Листинг 17.19. `binary.cpp`

```
// binary.cpp -- бинарный файловый ввод-вывод
#include <iostream>      // для большинства систем не требуется
#include <fstream>
#include <iomanip>
#include <cstdlib>       // для exit()

inline void eatline() { while (std::cin.get() != '\n') continue; }
struct planet
{
   char name[20];        // название планеты
   double population;    // население
   double g;             // ускорение свободного падения
};

const char * file = "planets.dat";

int main()
{
   using namespace std;
   planet pl;
   cout << fixed << right;

   // Отображение начального содержимого
   ifstream fin;
   fin.open(file, ios_base::in |ios_base::binary);   // бинарный файл
   // Примечание: некоторые системы не принимают режим ios_base::binary
   if (fin.is_open())
   {
      cout << "Here are the current contents of the "
         << file << " file:\n";
```

```
    while (fin.read((char *) &pl, sizeof pl))
    {
        cout << setw(20) << pl.name << ": "
             << setprecision(0) << setw(12) << pl.population
             << setprecision(2) << setw(6) << pl.g << endl;
    }
    fin.close();
}
// Добавление новых данных
ofstream fout(file,
       ios_base::out | ios_base::app | ios_base::binary);
// Примечание: некоторые системы не принимают режим ios::binary
if (!fout.is_open())
{
    cerr << "Can't open " << file << " file for output:\n";
    exit(EXIT_FAILURE);
}
cout << "Enter planet name (enter a blank line to quit):\n"; // ввод названия планеты
cin.get(pl.name, 20);
while (pl.name[0] != '\0')
{
    eatline();
    cout << "Enter planetary population: ";                  // ввод населения
    cin >> pl.population;
    cout << "Enter planet's acceleration of gravity: ";
         // Ввод ускорения свободного падения
    cin >> pl.g;
    eatline();
    fout.write((char *) &pl, sizeof pl);
    cout << "Enter planet name (enter a blank line "
            "to quit):\n";                                   // ввод названия планеты
    cin.get(pl.name, 20);
}
fout.close();
// Отображение измененного файла
fin.clear();  // не обязательно в некоторых реализациях, но это не помешает
fin.open(file, ios_base::in | ios_base::binary);
if (fin.is_open())
{
    cout << "Here are the new contents of the "
         << file << " file:\n";
    while (fin.read((char *) &pl, sizeof pl))
    {
        cout << setw(20) << pl.name << ": "
             << setprecision(0) << setw(12) << pl.population
             << setprecision(2) << setw(6) << pl.g << endl;
    }
    fin.close();
}
cout << "Done.\n";
return 0;
}
```

Ниже приведен пример первоначального выполнения программы из листинга 17.19:

```
Enter planet name (enter a blank line to quit):
Earth
Enter planetary population: 6928198253
Enter planet's acceleration of gravity: 9.81
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat file:
              Earth:   6928198253  9.81
Done.
```

А это — пример следующего ее запуска:

```
Here are the current contents of the planets.dat file:
              Earth:   6928198253  9.81
Enter planet name (enter a blank line to quit):
Jenny's World
Enter planetary population: 32155648
Enter planet's acceleration of gravity: 8.93
Enter planet name (enter a blank line to quit):

Here are the new contents of the planets.dat file:
              Earth:   6928198253  9.81
      Jenny's World:     32155648  8.93
Done.
```

Вы уже видели основные возможности этой программы, но задержимся на ней еще немного. Программа использует следующий код (в форме встроенной функции `eatlene()`) после чтения значения `g` для планеты:

```
while (std::cin.get() != '\n') continue;
```

Он считывает и отбрасывает ввод вплоть до символа новой строки. Рассмотрим следующий оператор ввода в цикле:

```
cin.get(pl.name, 20);
```

Если бы символ новой строки остался на своем месте, этот оператор читал бы его как пустую строку, завершая цикл.

Вы можете спросить: нельзя ли воспользоваться объектом `string` вместо массива символов для члена `name` структуры `planet`? Ответ отрицателен — по крайней мере, без серьезных изменений в проектном решении. Проблема в том, что объект `string` в действительности не содержит в себе строку. Вместо этого он содержит указатель на область памяти, где хранится строка. Поэтому если скопировать структуру в файл, то будут скопированы не данные строки, а адрес области ее хранения в памяти. При повторном запуске программы этот адрес утрачивает смысл.

Произвольный доступ

В качестве последнего примера обработки файла рассмотрим применение произвольного доступа. *Произвольный доступ* означает возможность перемещения в любую позицию в файле вместо последовательного перемещения по нему. Подход с произвольным доступом часто применяется при работе с файлами баз данных. Программа будет поддерживать отдельный индексный файл с информацией о местоположении данных в основном файле. В этом случае она сможет "перепрыгивать" непосредственно в заданное место, читать там данные и, возможно, модифицировать их. Этот подход проще всего реализовать, если файл состоит из набора записей одинаковой длины. Каждая запись представляет связанный набор данных. Например, в примере, приведенном в листинге 17.19, каждая запись файла будет представлять всю информа-

цию об определенной планете. Запись в файле достаточно естественно соответствует структуре программы или классу.

Этот пример основан на программе работы с бинарным файлом из листинга 17.19. При этом программа пользуется преимуществом того, что структура `planet` предоставляет шаблон для записи файла. Чтобы придать программированию творческий характер, этот пример открывает файл в режиме чтения-записи, предоставляя возможность как чтения, так и модификации записей. Это можно реализовать, создав объект `fstream`. Класс `fstream` является производным от класса `iostream`, который, в свою очередь, базируется на классах `istream` и `ostream`, а потому наследует методы обоих этих классов. Он также наследует два буфера — один для ввода и один для вывода — и синхронизирует управление этим двумя буферами. Таким образом, как только программа считывает файл или записывает в него, она одновременно перемещает указатель ввода в буфере ввода и указатель вывода в буфере вывода.

Пример выполняет следующие действия.

1. Отображает текущее содержимое файла `planets.dat`.
2. Запрашивает, какую запись требуется модифицировать.
3. Модифицирует эту запись.
4. Отображает измененное содержимое файла.

Более претенциозная программа может использовать меню и цикл, чтобы предоставить возможность выбора из списка доступных действий неограниченное количество раз, но эта версия выполняет каждое действие лишь однажды. Этот упрощенный подход позволяет исследовать несколько аспектов чтения-записи файлов, не слишком погружаясь в проблемы дизайна программы.

Внимание!

Эта программа исходит из предположения, что файл `planets.dat` уже существует, и он был создан программой `binary.cpp` из листинга 17.19.

Первый вопрос, на который следует ответить — какой режим файла использовать? Чтобы файл был доступен для чтения, требуется режим `ios_base::in`. Для бинарного ввода-вывода необходим режим `ios_base::binary`. (Повторимся еще раз: в некоторых нестандартных системах можно, и часто даже нужно, опускать этот режим.) Чтобы иметь возможность записи в файл, требуется режим `ios_base::out` или `ios_base::app`. Однако режим дополнения позволяет программе добавлять данные только в конец файла. Остальная часть файла остается доступной только для чтения; т.е. можно считывать исходные данные, но не модифицировать их — поэтому нужно использовать режим `ios_base::out`. Как показано в табл. 17.8, совместное применение режимов `in` и `out` обеспечивает режим чтения-записи, поэтому нужно только добавить элемент `binary`. Как уже упоминалось, для объединения режимов служит операция `|`. Таким образом, чтобы подготовить файл к обработке, нужно использовать следующий оператор:

```
finout.open(file,ios_base::in | ios_base::out | ios_base::binary);
```

Далее понадобится способ перемещения по файлу. Класс `fstream` наследует два предназначенных для этого метода: `seekg()` перемещает в заданную позицию файла указатель ввода, а `seekp()` перемещает указатель вывода. (На самом деле, поскольку класс `fstream` использует буферы для промежуточного хранения данных, эти указатели указывают на положение в буферах, а не в реальном файле.)

Можно также применять seekg() с объектом ifstream, а seekp() — с объектом ofstream. Прототипы seekg() имеют следующий вид:

```
basic_istream<charT,traits>& seekg(off_type, ios_base::seekdir);
basic_istream<charT,traits>& seekg(pos_type);
```

Как видите, они представляют собой шаблоны. В этой главе используется специализация шаблона для типа char. Для такой специализации приведенные два прототипа эквивалентны следующим:

```
istream & seekg(streamoff, ios_base::seekdir);
istream & seekg(streampos);
```

Первый прототип представляет отыскание позиции в файле, измеренной как смещение в байтах от позиции, заданной вторым аргументом. Второй прототип представляет отыскание позиции в файле, измеренной в байтах от начала файла.

Повышение типа

Когда C++ был молодым языком, для методов seekg() жизнь была проще. Типы streamoff и streampos были определены как typedef для некоторого целочисленного типа, такого как long. Однако решение задачи создания переносимого стандарта привело к осознанию того, что целочисленный аргумент может предоставлять недостаточно информации для некоторых файловых систем, поэтому streamoff и streampos было разрешено быть структурами или типами классов — лишь бы они допускали выполнение некоторых базовых операций вроде использования целочисленного значения в качестве инициализирующего. Затем старый класс istream был заменен шаблоном basic_istream, а streamoff и streampos — шаблонными типами pos_type и off_type. Однако streampos и streamoff продолжают существовать в качестве char-специализаций типов pos_type и off_type. Аналогично, теперь можно использовать типы wstreamoff и wstreampos, если seekg() применяется с объектом типа wistream.

Взгляните на аргументы первого прототипа seekg(). Значения типа streamoff используются для измерения смещений в байтах от определенного положения в файле. Аргумент streamoff представляет позицию в файле в байтах, измеренную как смещение от одного из трех возможных положений. (Тип может быть определен как целочисленный или класс.)

Аргумент seek_dir — еще один целочисленный тип, который определен вместе с тремя возможными значениями в классе ios_base. Константа ios_base::beg означает отсчет смещения от начала файла, константа ios_base::cur — от текущей позиции, а константа ios_base::end — от конца файла. Вот некоторые примеры вызовов, предполагающие, что fin — объект класса ifstream:

```
fin.seekg(30, ios_base::beg);      // 30 байт от начала
fin.seekg(-1, ios_base::cur);      // назад на 1 байт
fin.seekg(0, ios_base::end);       // перейти в конец файла
```

Теперь взгляните на второй прототип. Значения типа streampos определяют позицию в файле. Это может быть класс, который должен включать конструктор с аргументом streamoff и конструктор с целочисленным аргументом, предоставляя способ преобразования обоих типов в значения streampos. Значение streampos представляет абсолютную позицию в файле, измеренную от его начала. Позицию streampos можно трактовать, как если бы она измеряла положение в файле в байтах от начала файла, причем первым байтом был бы нулевой. Таким образом, следующий оператор устанавливает указатель файла на 112-й байт, который в файле является 113-м байтом:

```
fin.seekg(112);
```

Если требуется проверить текущую позицию файлового указателя, можно воспользоваться методом `tellg()` для входных потоков и методами `tellp()` — для выходных. Каждый из них возвращает значение `streampos`, представляющее текущую позицию в байтах, измеренную от начала файла. При создании объекта `fstream` указатели ввода и вывода перемещаются в тандеме, поэтому `tellg()` и `tellp()` возвращают одно и то же значение. Но если вы применяете объект `istream` для управления входным потоком и объект `ostream` для управления выходным потоком в одном и том же файле, указатели ввода и вывода перемещаются независимо друг от друга, поэтому `tellg()` и `tellp()` могут возвращать разные значения. С помощью метода `seekg()` можно перемещаться к началу файла. Ниже приведен фрагмент кода, который открывает файл, переходит в его начало и отображает содержимое:

```
fstream finout;           // потоки чтения и записи
finout.open(file,ios::in | ios::out |ios::binary);
// Примечание: некоторые системы Unix требуют опустить | ios::binary
int ct = 0;
if (finout.is_open())
{
   finout.seekg(0);       // перейти в начало
   cout << "Here are the current contents of the "
        << file << " file:\n";
   while (finout.read((char *) &pl, sizeof pl))
   {
     cout << ct++ << ": " << setw(LIM) << pl.name << ": "
          << setprecision(0) << setw(12) << pl.population
          << setprecision(2) << setw(6) << pl.g << endl;
   }
   if (finout.eof())
      finout.clear();    // очистить флаг eof
   else
   {
     cerr << "Error in reading " << file << ".\n"; // ошибка при чтении файла
     exit(EXIT_FAILURE);
   }
}
else
{
   cerr << file << " could not be opened -- bye.\n";
   exit(EXIT_FAILURE);
}
```

Этот фрагмент похож на начало листинга 17.19, но с некоторыми изменениями и дополнениями. Как только что было описано, программа использует объект `fstream` с режимом чтения-записи, и применяет `seekg()` для позиционирования файлового указателя в начало файла. (В действительности для данного примера это не требуется, но позволяет продемонстрировать использование `seekg()`.) Далее программа вносит небольшое изменение в нумерации отображаемых записей. После этого вносится следующее важное дополнение:

```
if (finout.eof())
   finout.clear();        // очистить флаг eof
else
{
   cerr << "Error in reading " << file << ".\n";
   exit(EXIT_FAILURE);
}
```

Проблема в том, что когда программа заканчивает чтение и отображение всего файла, она устанавливает элемент eofbit. Это доказывает, что работа с файлом завершена, и любые дальнейшие операции чтения или записи в файл становятся невозможными. Применение метода clear() сбрасывает состояние потока, очищая eofbit. Теперь программа может снова обратиться к файлу. Часть else обрабатывает возможную ситуацию, когда программа прекращает чтение файла по какой-либо иной причине, чем достижение конца файла – например, по причине аппаратного сбоя.

Следующий шаг состоит в идентификации записи, которая должна быть изменена, и последующем ее изменении. Для этого программа запрашивает у пользователя номер записи. Умножение введенного номера на размер записи в байтах дает номер байта, с которого начинается искомая запись. Если record – номер записи, то требуемым номером байта будет record * sizeof pl:

```
cout << "Enter the record number you wish to change: ";
long rec;
cin >> rec;
eatline();                          // избавление от символов новой строки
if (rec < 0 || rec >= ct)
{
   cerr << "Invalid record number -- bye\n";
   exit(EXIT_FAILURE);
}
streampos place = rec * sizeof pl;  // преобразование в тип streampos
finout.seekg(place);                // произвольный доступ
```

Переменная ct представляет количество записей; при попытке выхода за пределы файла программа завершается.

Затем программа отображает текущую запись:

```
finout.read((char *) &pl, sizeof pl);
cout << "Your selection:\n";
cout << rec << ": " << setw(LIM) << pl.name << ": "
     << setprecision(0) << setw(12) << pl.population
     << setprecision(2) << setw(6) << pl.g << endl;
if (finout.eof())
   finout.clear();                  // очистить флаг eof
```

После отображения записи программа позволяет ее изменить:

```
cout << "Enter planet name: ";
cin.get(pl.name, LIM);
eatline();
cout << "Enter planetary population: ";
cin >> pl.population;
cout << "Enter planet's acceleration of gravity: ";
cin >> pl.g;
finout.seekp(place);                // вернуться назад
finout.write((char *) &pl, sizeof pl) << flush;
if (finout.fail())
{
   cerr << "Error on attempted write\n";
   exit(EXIT_FAILURE);
}
```

Программа сбрасывает вывод, чтобы гарантировать обновление файла перед тем, как перейти к следующей стадии обработки.

И, наконец, чтобы отобразить измененное содержимое файла, программа применяет seekg(), чтобы сбросить указатель файла на начало. Полный код программы приведен в листинге 17.20. Не забудьте, что в ней предполагается, что файл planets.dat, созданный с помощью программы binary.cpp, доступен.

На заметку!

Чем старше реализация, тем более вероятно, что она не соответствует стандарту C++. Некоторые системы не распознают флаг binary, манипуляторы fixed и right, а также ios_base.

Листинг 17.20. random.cpp

```
// random.cpp -- произвольный доступ к бинарному файлу
#include <iostream>           // для большинства систем не требуется
#include <fstream>
#include <iomanip>
#include <cstdlib>            // для exit()
const int LIM = 20;
struct planet
{
   char name[LIM];            // название планеты
   double population;         // население
   double g;                  // ускорение свободного падения
};

const char * file = "planets.dat"; // ПРЕДПОЛАГАЕТСЯ, ЧТО СУЩЕСТВУЕТ (пример binary.cpp)
inline void eatline() { while (std::cin.get() != '\n') continue; }

int main()
{
   using namespace std;
   planet pl;
   cout << fixed;

   // Отображение начального содержимого
   fstream finout;            // потоки чтения и записи
   finout.open(file,
        ios_base::in | ios_base::out | ios_base::binary);
   // Примечание: некоторые системы Unix требуют опустить | ios::binary
   int ct = 0;
   if (finout.is_open())
   {
      finout.seekg(0);        // перейти в начало
      cout << "Here are the current contents of the "
           << file << " file:\n";
      while (finout.read((char *) &pl, sizeof pl))
      {
         cout << ct++ << ": " << setw(LIM) << pl.name << ": "
              << setprecision(0) << setw(12) << pl.population
             << setprecision(2) << setw(6) << pl.g << endl;
      }
      if (finout.eof())
         finout.clear();      // очистить флаг eof
      else
      {
         cerr << "Error in reading " << file << ".\n";
         exit(EXIT_FAILURE);
      }
   }
```

```
    else
    {
        cerr << file << " could not be opened -- bye.\n";
        exit(EXIT_FAILURE);
    }
    // Изменить запись
    cout << "Enter the record number you wish to change: ";
    long rec;
    cin >> rec;
    eatline();                          // избавление от символов новой строки
    if (rec < 0 || rec >= ct)
    {
        cerr << "Invalid record number -- bye\n";
        exit(EXIT_FAILURE);
    }
    streampos place = rec * sizeof pl;  // преобразование в тип streampos
    finout.seekg(place);                // произвольный доступ
    if (finout.fail())
    {
        cerr << "Error on attempted seek\n";
        exit(EXIT_FAILURE);
    }
    finout.read((char *) &pl, sizeof pl);
    cout << "Your selection:\n";
    cout << rec << ": " << setw(LIM) << pl.name << ": "
         << setprecision(0) << setw(12) << pl.population
         << setprecision(2) << setw(6) << pl.g << endl;
    if (finout.eof())
        finout.clear();                 // очистить флаг eof
    cout << "Enter planet name: ";
    cin.get(pl.name, LIM);
    eatline();
    cout << "Enter planetary population: ";
    cin >> pl.population;
    cout << "Enter planet's acceleration of gravity: ";
    cin >> pl.g;
    finout.seekp(place);                // вернуться назад
    finout.write((char *) &pl, sizeof pl) << flush;
    if (finout.fail())
    {
        cerr << "Error on attempted write\n";
        exit(EXIT_FAILURE);
    }
    // Отображение измененного файла
    ct = 0;
    finout.seekg(0);                    // перейти в начало файла
    cout << "Here are the new contents of the " << file
         << " file:\n";
    while (finout.read((char *) &pl, sizeof pl))
    {
        cout << ct++ << ": " << setw(LIM) << pl.name << ": "
             << setprecision(0) << setw(12) << pl.population
             << setprecision(2) << setw(6) << pl.g << endl;
    }
    finout.close();
    cout << "Done.\n";
    return 0;
}
```

Ниже представлен пример запуска программы из листинга 17.20 на основе файла planets.dat, в который было добавлено несколько новых записей с тех пор, как мы его видели последний раз:

```
Here are the current contents of the planets.dat file:
0:             Earth:   6928198253  9.81
1:       Jenny's World:     32155648  8.93
2:            Tramtor:  89000000000 15.03
3:            Trellan:      5214000  9.62
4:          Freestone:   3945851000  8.68
5:          Taanagoot:    361000004 10.23
6:             Marin:       252409  9.79
Enter the record number you wish to change: 2
Your selection:
2:            Tramtor:  89000000000 15.03
Enter planet name: Trantor
Enter planetary population: 89521844777
Enter planet's acceleration of gravity: 10.53
Here are the new contents of the planets.dat file:
0:             Earth:   6928198253  9.81
1:       Jenny's World:     32155648  8.93
2:            Trantor:  89521844777 10.53
3:            Trellan:      5214000  9.62
4:          Freestone:   3945851000  8.68
5:          Taanagoot:    361000004 10.23
6:             Marin:       252409  9.79
Done.
```

Используя методики, представленные в этой программе, ее можно расширить так, чтобы она позволяла добавлять новую информацию и удалять записи. Если вы намерены расширить программу, целесообразно реорганизовать ее, применив классы и функции. Например, структуру planet можно было бы преобразовать в определение класса, затем перегрузить операцию вставки << так, чтобы выражение cout << pl отображало члены данных класса сформатированными, как в приведенном примере. К тому же приведенный пример не заботится о проверке допустимости ввода, поэтому можно было бы добавить код, проверяющий допустимость вводимых числовых данных, когда это необходимо.

Работа с временными файлами

При разработке приложений часто требуется использовать временные файлы, время жизни которых непродолжительно и должно управляться программой. Думали ли вы когда-нибудь о том, как это реализовать на C++? В действительности создать временный файл, скопировать его содержимое в другой файл и уничтожить его достаточно просто. Прежде всего, нужно продумать схему именования своих временных файлов. Но как можно гарантировать, что каждому из них будет назначено уникальное имя? На помощь приходит стандартная функция tmpnam(), объявленная в cstdio:

```
char* tmpnam( char* pszName );
```

Функция tmpnam() создает временное имя и помещает его в строку в стиле C, на которую указывает pszName. Обе константы L_tmpnam и TMP_MAX, определенные в cstdio, ограничивают количество символов в имени файла и максимальное число вызовов tmpnam() без генерации повторяющихся имен файлов в текущем каталоге. Следующий пример генерирует 10 временных имен:

```
#include <cstdio>
#include <iostream>
int main()
{
   using namespace std;
   cout << "This system can generate up to " << TMP_MAX
        << " temporary names of up to " << L_tmpnam
        << " characters.\n";
   char pszName[ L_tmpnam ] = {'\0'};
   cout << "Here are ten names:\n";
   for( int i=0; 10 > i; i++ )
   {
      tmpnam( pszName );
      cout << pszName << endl;
   }
   return 0;
}
```

В общем случае, используя `tmpnam()`, теперь можно генерировать `TMP_MAX` уникальных имен, каждое длиной до `L_tmpnam` символов. Сами имена зависят от компилятора. Чтобы посмотреть, какие имена сгенерирует используемый компилятор, можно запустить эту программу.

Внутреннее форматирование

Семейство `iostream` поддерживает выполнение ввода-вывода между программой и терминалом. Семейство `fstream` использует тот же интерфейс для обеспечения операций ввода-вывода между программой и файлом. Библиотека C++ также предоставляет семейство `sstream`, которое применяет тот же интерфейс для организации ввода-вывода между программой и объектом `string`. Это значит, что для записи форматированной информации в объект `string` можно использовать те же методы `ostream`, которые применялись с `cout`, а для чтения информации из объекта `string` — такие методы `istream`, как `getline()`. Процесс чтения форматированной информации из объекта `string` и записи форматированной информации в объект `string` называется *внутренним* форматированием. Давайте кратко рассмотрим эти возможности. (Семейство `sstream` поддержки `string` замещает семейство `strstream.h`, поддерживающее символьные массивы.)

Заголовочный файл `sstream` определяет класс `ostringstream`, который является производным от класса `ostream`. (Существует также класс `wostringstream`, базирующийся на `wostream`, который предназначен для расширенных символьных наборов.) Если создать объект `ostringstream`, в него можно записывать информацию, которая сохраняется. С объектом `ostringstream` можно применять те же методы, что и с объектом `cout`. Таким образом, можно написать примерно такой код:

```
ostringstream outstr;
double price = 380.0;
char * ps = " for a copy of the ISO/EIC C++ standard!";
outstr.precision(2);
outstr << fixed;
outstr << "Pay only CHF " << price << ps << endl;
```

Форматированный текст помещается в буфер, и объект использует динамическое выделение памяти для расширения размера буфера при необходимости. Класс

ostringstream имеет функцию-член str(), которая возвращает объект string, инициализированный содержимым буфера:

```
string mesg = outstr.str(); // возвращает строку с форматированной информацией
```

Применение str() "замораживает" объект, и дальнейшая запись в него становится невозможной. В листинге 17.21 приведен краткий пример внутреннего форматирования.

Листинг 17.21. strout.cpp

```
// strout.cpp — внутреннее форматирование (вывод)
#include <iostream>
#include <sstream>
#include <string>
int main()
{
   using namespace std;
   ostringstream outstr;           // управляет строковым потоком
   string hdisk;
   cout << "What's the name of your hard disk? ";
   getline(cin, hdisk);
   int cap;
   cout << "What's its capacity in GB? ";
   cin >> cap;
   // Запись форматированной информации в строковый поток
   outstr << "The hard disk " << hdisk << " has a capacity of "
          << cap << " gigabytes.\n";
   string result = outstr.str();   // сохранение результата
   cout << result;                 // отображение содержимого
   return 0;
}
```

Ниже показан пример выполнения программы из листинга 17.21:

```
What's the name of your hard disk? Datarapture
What's its capacity in GB? 2000
The hard disk Datarapture has a capacity of 2000 gigabytes.
```

Класс istringstream позволяет использовать семейство методов istream для чтения данных из объекта istringstream, который может быть инициализирован объектом string.

Предположим, что facts — это объект типа string. Чтобы создать объект istringstream, ассоциированный с этой строкой, можно использовать следующий код:

```
istringstream instr(facts);  // использование facts для инициализации потока
```

Затем можно применить методы istream для чтения данных из instr. Например, если instr содержит ряд целых чисел в символьном формате, их можно прочесть следующим образом:

```
int n;
int sum = 0;
while (instr >> n)
   sum += n;
```

В листинге 17.22 применяется перегруженная операция >> для чтения содержимого строки по одному слову за раз.

Листинг 17.22. strin.cpp

```
// strin.cpp -- форматированное чтение из символьного массива
#include <iostream>
#include <sstream>
#include <string>
int main()
{
   using namespace std;
   string lit = "It was a dark and stormy day, and "
            " the full moon glowed brilliantly. ";
   istringstream instr(lit);       // использование буфера для ввода
   string word;
   while (instr >> word)           // чтение по одному слову
      cout << word << endl;
   return 0;
}
```

Вывод программы из листинга 17.22 имеет следующий вид:

```
It
was
a
dark
and
stormy
day,
and
the
full
moon
glowed
brilliantly.
```

Короче говоря, классы `istringstream` и `istringstream` предоставляют в ваше распоряжение всю мощь методов классов `istream` и `ostream` для управления символьными данными, сохраненными в строках.

Резюме

Поток — это последовательность байтов, передаваемых в программу или из нее. Буфер — область памяти для временного хранения, служащая посредником между программой и файлом или другими устройствами ввода-вывода. Информация может передаваться между буфером и файлом большими порциями данных, размер которых позволяет наиболее эффективно работать с такими устройствами, как дисководы. Информация может также передаваться между буфером и программой байт за байтом, что часто значительно удобнее для обработки в программе. В C++ ввод обрабатывается за счет подключения буферизованного потока к программе и источнику данных. Аналогично, в C++ вывод обрабатывается подключением буферизованного потока к программе и целевому устройству или файлу. Файлы `iostream` и `fstream` определяют библиотеку классов ввода-вывода, включающую богатый набор классов управления потоками. Программы C++, которые включают файл `iostream`, автоматически открывают восемь потоков, управляя ими посредством восьми объектов. Объект `cin` управляет стандартным потоком ввода, который по умолчанию подключен к стандартному устройству ввода — как правило, клавиатуре. Объект `cout` управляет стандартным вы-

ходным потоком, который по умолчанию подключен к стандартному устройству вывода – обычно монитору. Объекты `cerr` и `clog` управляют, соотвстственно, небуферизованным и буферизованным потоками, подключенными к стандартному устройству ошибок – как правило, монитору. Эти четыре объекта имеют четыре аналога, работающие с широкими символами, а именно: `wcin`, `wcout`, `wcerr` и `wclog`.

Библиотека классов ввода-вывода предоставляет широкий набор удобных методов. Класс `istream` определяет версии операций извлечения (>>), которые распознают все базовые типы C++ и преобразуют символьный ввод в эти типы. Семейство методов `get()` и метод `getline()` обеспечивают дополнительную поддержку односимвольного и строкового ввода. Аналогично, класс `ostream` определяет версии операций вставки (<<), которые распознают все базовые типы C++ и преобразуют их в соответствующий символьный вывод. Метод `put()` предлагает дополнительную поддержку односимвольного вывода. Классы `wistream` и `wostream` предоставляют аналогичную поддержку "широких" символов.

Тем, как программа форматирует вывод, можно управлять, используя методы класса `ios_base` и применяя манипуляторы (функции, которые могут быть конкатенированы с операциями вставки), определенные в файлах `iostream` и `iomanip`. Эти методы и манипуляторы позволяют управлять основанием системы счисления, шириной поля, количеством отображаемых десятичных разрядов при выводе значений с плавающей точкой, а также другими элементами.

Файл `fstream` предоставляет определения классов, которые расширяют методы `iostream` для файлового ввода-вывода. Класс `ifstream` является производным от класса `istream`. Ассоциируя объект `ifstream` с файлом, методы `iostream` можно использовать для чтения файла. Аналогично, ассоциация объекта `ofstream` с файлом позволяет применять методы `ostream` для записи в файл. А вот ассоциация объекта `fstream` с файлом позволяет использовать с файлом и методы ввода, и методы вывода.

Чтобы ассоциировать файл с потоком, можно указать имя файла при инициализации объекта файлового потока либо сначала создать объект файлового потока, а затем применить метод `open()` для его ассоциирования с файлом. Метод `close()` разрывает связь между потоком и файлом. Конструкторы классов и метод `open()` принимают не обязательный второй аргумент, указывающий режим файла. Режим файла определяет такие аспекты, как возможность чтения и/или записи файла, должно ли происходить усечение файла при открытии его для записи, будет ли считаться ошибкой попытка открытия несуществующего файла, а также то, какой режим нужно использовать – бинарный или текстовый.

Текстовый файл хранит всю информацию в символьной форме. Например, числовые значения преобразуются в символьные представления. Обычные операции вставки и извлечения, а также методы `get()` и `getline()`, поддерживают этот режим. Бинарный файл сохраняет всю информацию, используя то же бинарное представление, которое внутренне применяет компьютер. Бинарные файлы хранят данные – в частности, значения с плавающей точкой – более точно и компактно, чем текстовые файлы, но при этом они менее переносимы. Методы `read()` и `write()` поддерживают бинарный ввод и вывод.

Функции `seekg()` и `seekp()` предоставляют программам C++ произвольный доступ к файлам. Эти методы класса позволяют позиционировать указатель файла относительно начала файла, его конца или текущей позиции. Методы `tellg()` и `tellp()` сообщают текущую позицию указателя в файле.

Заголовочный файл `sstream` определяет классы `istringstream` и `ostringstream`, которые позволяют использовать методы `istream` и `ostream` для извлечения информации из строки и форматирования информации, помещаемой в строку.

Вопросы для самопроверки

1. Какую роль играет файл iostream во вводе-выводе C++?
2. Почему ввод числа 121 с клавиатуры требует от программы выполнения преобразования?
3. В чем состоит различие между потоком стандартного вывода и стандартным потоком ошибок?
4. Почему cout может отображать различные типы C++ без необходимости явного указания инструкций для каждого типа?
5. Какое свойство определений методов вывода позволяет выполнять конкатенацию вывода?
6. Напишите программу, которая запрашивает целое число и затем отображает его в десятичной, восьмеричной и шестнадцатеричной формах. Отобразите все формы в одной и той же строке, в полях шириной по 15 символов, с применением префиксов C++ для оснований систем счисления.
7. Напишите программу, которая запрашивает следующую информацию и форматирует ее, как показано ниже:

```
Enter your name: Billy Gruff
Enter your hourly wages: 12
Enter number of hours worked: 7.5
First format:
               Billy Gruff: $    12.00:  7.5
Second format:
Billy Gruff                 : $12.00    :7.5
```

8. Пусть имеется следующая программа:

```
//rq17-8.cpp
#include <iostream>
int main()
{
   using namespace std;
   char ch;
   int ct1 = 0;

   cin >> ch;
   while (ch != 'q')
   {
      ct1++;
      cin >> ch;
   }

   int ct2 = 0;
   cin.get(ch);
   while (ch != 'q')
   {
      ct2++;
      cin.get(ch);
   }
   cout << "ct1 = " << ct1 << "; ct2 = " << ct2 << "\n";

   return 0;
}
```

Что она напечатает, если получит следующий ввод:

```
I see a q<Enter>
I see a q<Enter>
```

Здесь <Enter> означает нажатие одноименной клавиши.

9. Оба следующих оператора читают и отбрасывают символы, вплоть до конца строки, включая его. Чем различается их поведение?

```
while (cin.get() != '\n')
   continue;
cin.ignore(80, '\n');
```

Упражнения по программированию

1. Напишите программу, которая подсчитывает количество символов вплоть до первого символа $ в строке, оставляя $ во входном потоке.
2. Напишите программу, которая копирует клавиатурный ввод (вплоть до эмулируемого конца файла) в файл, имя которого передано в командной строке.
3. Напишите программу, копирующую один файл в другой. Имена файлов программа должна получать из командной строки. Если не удается открыть файл, должно выдаваться соответствующее сообщение.
4. Напишите программу, которая открывает два текстовых файла для ввода и один для вывода. Программа должна соединять соответствующие строки входных файлов, используя в качестве разделителя пробел, и записывать результаты в выходной файл. Если один файл короче второго, остальные строки более длинного файла также должны копироваться в выходной файл. Например, предположим, что первый входной файл имеет следующее содержимое:

```
eggs kites donuts
balloons hammers
stones
```

А второй файл — такое:

```
zero lassitude
finance drama
```

Результирующий файл должен выглядеть следующим образом:

```
eggs kites donuts zero lassitude
balloons hammers finance drama
stones
```

5. Мэт (Mat) и Пэт (Pat) хотят пригласить своих друзей на вечеринку — почти так же, как они это делали в упражнении 8 из главы 16, за исключением того, что теперь им нужна программа, использующая файлы. Они просят вас написать программу, которая должна выполнять перечисленные ниже действия.
   - Считывать список друзей Мэта из текстового файла `mat.dat`, в котором в каждой строке указан один друг. Имена сохраняются в контейнере и затем отображаются в отсортированном виде.
   - Считывать список друзей Пэт из текстового файла `pat.dat`, в котором в каждой строке указан один друг. Имена сохраняются в контейнере и затем отображаются в отсортированном виде.
   - Объединить эти два списка, исключая дубликаты, и сохранить результат в файле `matnpat.dat`, по одному другу в строке.

6. Рассмотрите определение класса, предложенное в упражнении 5 главы 14. Если вы еще не делали это упражнение, выполните его сейчас. Затем сделайте следующее.

   Напишите программу, которая использует стандартный ввод-вывод C++ и файловый ввод-вывод в сочетании с данными типов `employee`, `manager`, `fink` и `highfink`, как определено в упражнении 5 из главы 14. Программа должна быть аналогична главным строкам листинга 17.17 в том, что должна позволять вносить новые данные в файл. При первом запуске программа должна запросить данные у пользователя, показать все введенные записи и сохранить информацию в файл. При последующих запусках она должна сначала прочитать и отобразить данные файла, дать возможность пользователю добавить новые данные и отобразить все данные снова. Единственное отличие должно состоять в том, что данные должны быть представлены в виде массива указателей на тип `employee`. Таким образом, указатель может указывать на объект `employee` либо на объект любого из трех производных от него типов. Сохраняйте массив маленьким, чтобы облегчить его проверку программой; например, его размер можно ограничить 10 элементами:

```
const int MAX = 10;    // не более 10 объектов
...
employee * pc[MAX];
```

   Для клавиатурного ввода программа должна использовать меню, чтобы предоставить пользователю выбор типа создаваемого объекта. С меню должен быть связан оператор `switch`, позволяющий использовать операцию `new` для создания объекта требуемого типа и присваивать адрес этого объекта указателю в массиве `pc`. Затем этот объект может вызвать виртуальную функцию `setall()` для запроса соответствующих данных от пользователя:

```
pc[i]->setall();  // вызывает функцию, соответствующую типу объекта
```

   Чтобы обеспечить сохранение данных в файле, необходимо объявить виртуальную функцию `writeall()`:

```
for (i = 0; i < index; i++)
   pc[i]->writeall(fout);  // объект fout типа ofstream
                           // подключен к выходному файлу
```

На заметку!

В упражнении 6 используйте текстовый, а не бинарный ввод-вывод. (К сожалению, виртуальные объекты содержат указатели на таблицы указателей на виртуальные функции, и `write()` копирует эту информацию в файл. Объект, заполняемый методом `read()` из файла, получает некорректные значения указателей на функции, что приводит к путанице в поведении виртуальных функций.) Используйте символы новой строки для отделения каждого поля данных от следующего — это облегчит идентификацию полей при вводе. Либо можно все же применить бинарный ввод-вывод, но не записывать объекты как единое целое. Вместо этого можно предусмотреть методы класса, которые применяют функции `read()` и `write()` к каждому отдельному члену класса, а не к объекту в целом. Таким образом, программа сможет записывать в файл только необходимые данные.

   Сложность представляет восстановление данных из файла. Проблема состоит в том, как программа узнает, какого типа объект будет восстановлен следующим: `employee`, `manager`, `fink` либо `highfink`? Один из возможных подходов к ре-

шению этой проблемы заключается в следующем: при записи данных объекта в файл предварить его целым числом, идентифицирующим тип следующего объекта. Затем при вводе из файла программа может читать это целое число и применять switch для создания объекта соответствующего типа для приема данных:

```
enum classkind{Employee, Manager, Fink, Highfink};  // в заголовке класса
...
int classtype;
while((fin >> classtype).get(ch)){  // символ новой строки отделяет
                                    // целое число от данных
   switch(classtype) {
      case Employee : pc[i] = new employee;
                  break;
```

Затем можно использовать указатель, чтобы вызвать виртуальную функцию getall() для считывания информации:

```
pc[i++]->getall();
```

7. Ниже представлена часть программы, которая читает клавиатурный ввод в вектор объектов string, сохраняет строковое содержимое (не объекты!) в файле, а затем копирует содержимое файла обратно в вектор объектов string:

```
int main()
{
   using namespace std;
   vector<string> vostr;
   string temp;

   // Получить строки
   cout << "Enter strings (empty line to quit):\n"; // запрос на ввод строк
   while (getline(cin,temp) && temp[0] != '\0')
         vostr.push_back(temp);
   cout << "Here is your input.\n";                 // вывод введенных строк
   for_each(vostr.begin(), vostr.end(), ShowStr);

   // Сохранить в файле
   ofstream fout("strings.dat", ios_base::out | ios_base::binary);
   for_each(vostr.begin(), vostr.end(), Store(fout));
   fout.close();

   // Восстановить содержимое файла
   vector<string> vistr;
   ifstream fin("strings.dat", ios_base::in | ios_base::binary);
   if (!fin.is_open())
   {
      cerr << "Could not open file for input.\n";
           // не удается открыть файл для ввода
      exit(EXIT_FAILURE);
   }
   GetStrs(fin, vistr);
   cout << "\nHere are the strings read from the file:\n";
          // строки, прочитанные из файла
   for_each(vistr.begin(), vistr.end(), ShowStr);

   return 0;
}
```

Обратите внимание, что файл открывается в бинарном формате и требуется, чтобы ввод-вывод осуществлялся методами `read()` и `write()`. Остается сделать немного, как перечислено ниже.

- Написать функцию `void ShowStr(const string &)`, которая отображает объект `string` с последующим символом перевода строки.
- Написать функтор `Store`, который записывает строковую информацию в файл. Конструктор `Store` должен указывать объект `ifstream`, а перегруженная функция `operator()(const string &)` должна указывать строку, подлежащую записи. Приемлемый подход состоит в записи в файл сначала размера строки, а затем — ее содержимого. Например, если `len` содержит размер строки, можно было бы использовать следующие операторы:

```
os.write((char *)&len, sizeof(std::size_t));  // сохранить длину
os.write(s.data(), len);                      // сохранить символы
```

  Член `data()` возвращает указатель на массив, который содержит символы строки. Он подобен члену `c_str()`, за исключением того, что последний добавляет нулевой символ.
- Написать функцию `GetStrs()`, которая восстанавливает информацию из файла. Она может использовать `read()` для получения размера строки и затем применять цикл для чтения указанного количества символов из файла, добавляя их в изначально пустую временную строку. Поскольку данные объекта `string` — закрытые, для извлечения данных в строку должен использоваться метод класса вместо считывания их напрямую в нее.

18

Новый стандарт C++

В ЭТОЙ ГЛАВЕ...

- Семантика переноса и ссылки rvalue
- Лямбда-выражения
- Шаблон оболочки `function`
- Шаблоны с переменным числом аргументов

Основное внимание в этой главе сосредоточено на языковых изменениях, связанных с выходом C++11. Многие новые возможности C++11 в книге уже упоминались ранее, и мы начнем с их краткого обзора. Затем мы довольно детально рассмотрим дополнительные средства, после чего упомянем о некоторых дополнениях C++11, которые выходят за рамки тематики данной книги. (Учитывая, что проект C++11 длится на почти 80% дольше, чем C++98, покрыть абсолютно весь материал невозможно.) Наконец, мы кратко опишем библиотеку Boost.

Обзор уже известных функциональных средств C++11

К настоящему моменту имеет смысл освежить в памяти те изменения в C++11, которые рассматривались ранее в книге. Ниже приведен их краткий обзор.

Новые типы

В C++11 появились типы `long long` и `unsigned long long` для поддержки 64-битных (или шире) целых чисел, а также типы `char16_t` и `char32_t` для поддержки представлений 16- и 32-битных символов. Кроме того, добавились необработанные (raw) строки. Все эти нововведения обсуждались в главах 3 и 4.

Унифицированная инициализация

В C++11 расширена применимость списка в фигурных скобках (списковой инициализации), что позволяет его использовать со всеми встроенными типами, а также типами, определяемыми пользователем (т.е. объектами классов). Список может применяться как с, так и без знака =:

```
int x = {5};
double y {2.75};
short quar[5] {4,5,2,76,1};
```

Кроме того, синтаксис списковой инициализации может использоваться в новых выражениях:

```
int * ar = new int [4] {2,4,6,7};    // C++11
```

С объектами классов список в фигурных скобках может применяться вместо списка в круглых скобках для вызова конструктора:

```
class Stump
{
private:
   int roots;
   double weight;
public:
   Stump(int r, double w) : roots(r), weight(w) {}
};
Stump s1(3,15.6);                  // старый стиль
Stump s2{5, 43.4};                 // C++11
Stump s3 = {4, 32.1};              // C++11
```

Однако если класс имеет конструктор, аргумент которого является шаблоном `std::initializer_list`, то только этот конструктор может использовать форму списковой инициализации. Различные аспекты списковой инициализации обсуждались в главах 3, 4, 9, 10 и 16.

Сужение

Синтаксис списковой инициализации предоставляет защиту от сужения – т.е. против присваивания числового значения числовому типу, который недостаточно вместителен, чтобы хранить такое значение. Обычная инициализация позволяет предпринимать действия, которые могут иметь, а могут и не иметь смысла:

```
char c1 = 1.57e27;          // инициализация типа char значением double;
                            // поведение не определено
char c2 = 459585821;        // инициализация типа char значением int;
                            // поведение не определено
```

Однако в случае применения синтаксиса списковой инициализации компилятор не разрешает преобразования типов, при которых производится попытка сохранить значение в типе, который является более "узким", чем указанное значение:

```
char c1 {1.57e27};          // инициализация типа char значением double;
                            // ошибка времени компиляции
char c2 = {459585821};      // инициализация типа char значением int;
                            //выход за пределы диапазона, ошибка времени компиляции
```

В то же время преобразования в более широкие типы разрешены. Кроме того, преобразование в более узкий тип допускается, если значение находится в диапазоне, разрешенном для этого типа:

```
char c1 {66};               // инициализация типа char значением int; разрешена
double c2 = {66};           // инициализация типа double значением int; разрешена
```

`std::initializer_list`

C++11 предоставляет класс шаблона `initializer_list` (см. главу 16), который может использоваться в качестве аргумента конструктора. Если класс имеет подходящий конструктор, синтаксис фигурных скобок может применяться только с этим конструктором. Элементы в таком списке должны все принадлежать одному и тому же типу либо иметь возможность преобразования к одному и тому же типу. Контейнеры STL располагают конструкторами с аргументом `initializer_list`:

```
vector<int> a1(10);         // неинициализированный вектор с 10 элементами
vector<int> a2{10};         // список инициализаторов, a2 имеет 1 элемент,
                            // установленный в 10
vector<int> a3{4,6,1};      // 3 элемента, установленные в 4, 6, 1
```

Поддержку этого класса шаблона обеспечивает заголовочный файл `initializer_list`. Данный класс имеет функции-члены `begin()` и `end()`, задающие диапазон списка. Аргумент `initializer_list` можно использовать как в обычных функциях, так и в конструкторах:

```
#include <initializer_list>
double sum(std::initializer_list<double> il);
int main()
{
   double total = sum({2.5,3.1,4});     // 4 преобразуется в 4.0
   ...
}
double sum(std::initializer_list<double> il)
{
   double tot = 0;
   for (auto p = il.begin(); p != il.end(); p++)
      tot += *p;
   return tot;
}
```

Объявления

В C++11 реализовано множество средств, которые упрощают объявления, особенно в ситуациях, возникающих при использовании шаблонов.

auto

В C++11 ключевое слово auto лишено своего первоначального смысла в качестве спецификатора класса хранения (см. главу 9) и теперь применяется для реализации автоматического выведения типа при условии, что задан явный инициализатор (см. главу 3). Компилятор устанавливает тип переменной в тип инициализирующего значения:

```
auto maton = 112;          // maton получает тип int
auto pt = &maton;          // pt получает тип int *
double fm(double, int);
auto pf = fm;              // pf получает тип double (*)(double,int)
```

Ключевое слово auto может также упростить объявления шаблонов. Например, если il является объектом типа std::initializer_list<double>, следующий код

```
for (std::initializer_list<double>::iterator p = il.begin();
p !=il.end(); p++)
```

можно заменить таким:

```
for (auto p = il.begin(); p !=il.end(); p++)
```

decltype

Ключевое слово decltype создает переменную типа, который указан выражением. Приведенный ниже оператор означает "назначить y тот же самый тип, что x", где x представляет собой выражение:

```
decltype(x) y;
```

Вот еще пара примеров:

```
double x;

decltype(x*n) q;           // q получает тот же тип, что и x*n, т.е. double
decltype(&x) pd;           // pd получает тот же тип, что и &x, т.е. double *
```

Это особенно полезно в определениях шаблонов, когда тип может быть не определен вплоть до создания специфического экземпляра:

```
template<typename T, typename U)
void ef(T t, U u)
{
   decltype(T*U) tu;
   ...
}
```

Здесь tu — любой тип, полученный в результате выполнения операции T*U, при условии, что эта операция определена. Например, если T имеет тип char, а U — тип short, tu получит тип int, поскольку происходит автоматическое целочисленное расширение, принятое в целочисленной арифметике.

По сравнению с auto работа decltype является более сложной, и в зависимости от используемых выражений результирующие типы могут быть ссылками и иметь квалификаторы const. Ниже представлены другие примеры:

```
int j = 3;
int &k = j
const int &n = j;
decltype(n) i1;                    // i1 получает тип const int &
decltype(j) i2;                    // i2 получает тип int
decltype((j)) i3;                  // i3 получает тип int &
decltype(k + 1) i4;                // i4 получает тип int
```

Правила, которые приводят к получению таких результатов, описаны в главе 8.

Хвостовой возвращаемый тип

В C++11 появился новый синтаксис для объявления функций, при котором возвращаемый тип указывается после имени функции и списка параметров, а не перед ними:

```
double f1(double, int);            // традиционный синтаксис
auto f2(double, int) -> double;    // новый синтаксис,
                                   // возвращаемым типом является double
```

Новый синтаксис может выглядеть менее читабельным, чем традиционные объявления функций, однако он делает возможным использование `decltype` для указания возвращаемых типов шаблонных функций:

```
template<typename T, typename U)
auto eff(T t, U u) -> decltype(T*U)
{
   ...
}
```

Иллюстрируемая здесь проблема состоит в том, что когда компилятор читает список параметров `eff`, `T` и `U` не находятся в области видимости, поэтому любое использование `decltype` должно находиться после этого списка параметров. Новый синтаксис делает это возможным.

Псевдонимы шаблонов: `using =`

Было бы удобно создавать псевдонимы для длинных или сложных идентификаторов типов. В C++ уже имеется для этого `typedef`:

```
typedef std::vector<std::string>::iterator itType;
```

C++11 предоставляет второй синтаксис для создания псевдонимов (см. главу 14):

```
using itType = std::vector<std::string>::iterator;
```

Отличие в том, что этот новый синтаксис может применяться и для частичных специализаций шаблонов, тогда как `typedef` — нет:

```
template<typename T>
using arr12 = std::array<T,12>;    // шаблон для множества псевдонимов
```

Этот оператор специализирует шаблон `array<T, int>`, устанавливая параметр `int` в 12. Например, взгляните на следующие объявления:

```
std::array<double, 12> a1;
std::array<std::string, 12> a2;
They can be replaced with the following:
arr12<double> a1;
arr12(std::string> a2;
```

nullptr

Нулевой указатель — это такой указатель, который гарантированно не указывает на допустимые данные. Традиционно этот указатель в C++ представляется в исходном коде с помощью 0, несмотря на то, что внутреннее представление может быть другим. В результате возникает ряд проблем, поскольку 0 является и константой-указателем, и целочисленной константой. Как было показано в главе 12, для представления нулевого указателя в C++11 появилось ключевое слово nullptr; это тип указателя, который не может быть преобразован в целочисленный тип. С целью обратной совместимости в C++ по-прежнему разрешено использовать 0, и выражение nullptr == 0 вычисляется как true, однако применение nullptr вместо 0 обеспечивает лучшую безопасность типов. Например, функции, принимающей аргумент int, может быть передано значение 0, но попытку передачи такой функции указателя nullptr компилятор расценит как ошибку. Таким образом, для большей ясности и безопасности следует применять указатель nullptr, если компилятор его воспринимает.

Интеллектуальные указатели

Программа, которая использует new для выделения памяти из кучи (или свободного хранилища), должна применять delete для освобождения этой памяти, когда она больше не нужна. Ранее в C++ был введен интеллектуальный указатель auto_ptr, помогающий автоматизировать этот процесс. Последующий опыт программирования, особенно с использованием STL, показал, что требуется что-то более изощренное. Руководствуясь опытом программистов и решениями, предоставленными библиотекой BOOST, в C++11 тип auto_ptr был объявлен устаревшим, а вместо него введены новые типы интеллектуальных указателей: unique_ptr, shared_ptr и weak_ptr. Первые два из них рассматриваются в главе 16. Все новые интеллектуальные указатели спроектированы для работы с контейнерами STL и семантикой переноса.

Изменения в спецификации исключений

C++ предоставляет синтаксис для указания, какие исключения (если есть) функция может сгенерировать (см. главу 15):

```
void f501(int) throw(bad_dog);      // может сгенерировать исключение bad_dog
void f733(long long) throw();       // не генерирует исключений
```

Как и с auto_ptr, коллективный опыт программистского сообщества C++ показывал, что на практике спецификации исключений не работают настолько хорошо, как предполагалось. В результате стандарт C++11 объявил спецификации исключений устаревшими. Тем не менее, в комитете по стандартам посчитали важным документирование факта о том, что функция не генерирует исключений, поэтому для такой цели было добавлено ключевое слово noexcept:

```
void f875(short, short) noexcept;  // не генерирует исключений
```

Перечисления с областью видимости

Традиционные перечисления C++ предоставляют способ создания именованных констант. Однако они обеспечивают слишком низкий уровень контроля типов. Кроме того, областью видимости имен перечисления является область, в которой объявлено перечисление, а это означает, что два перечисления, определенные в одной и той же области видимости, не должны содержать члены с одинаковыми именами. Наконец, перечисления могут оказаться не полностью переносимыми, т.к. в различных реализациях для них могут быть выбраны разные лежащие в основе типы. В C++11 появился

вариант перечислений, которые решают упомянутую проблему. Новая форма отличается использованием в определении ключевого слова `class` или `struct`:

```
enum Old1 {yes, no, maybe};                              // традиционная форма
enum class New1 {never, sometimes, often, always};       // новая форма
enum struct New2 {never, lever, sever};                  // новая форма
```

За счет необходимости в явном указании области видимости эти новые формы позволяют избежать конфликтов имен. Таким образом, для идентификации этих примеров перечислений должны использоваться `New1::never` и `New2::never`. Дополнительные сведения можно найти в главе 10.

Изменения в классах

С целью упрощения и расширения классов в C++11 внесены многие изменения. Они включают разрешение конструкторам вызывать друг друга и быть унаследованными, улучшенные способы управления доступом к классам, а также конструкторы переноса и операции присваивания с переносом; все это будет рассматриваться в настоящей главе. А для начала имеет смысл ознакомиться с обзором изменений, которые обсуждались ранее.

Операции преобразования `explicit`

Одним из интересных аспектов ранних версий языка C++ была простота настройки автоматических преобразований для классов. По мере накопления опыта в программировании выяснилось, что автоматическое преобразование типов может приводить к проблемам в форме неожиданных преобразований. Один аспект этой проблемы был решен в C++ за счет ввода ключевого слова `explicit` для подавления автоматических преобразований, вызванных конструкторами с одним аргументом:

```
class Plebe
{
   Plebe(int);               // автоматическое преобразование int в Plebe
   explicit Plebe(double);   // требуется явное использование
   ...
};
...
Plebe a, b;
a = 5;                       // неявное преобразование, вызов Plebe(5)
b = 0.5;                     // не разрешено
b = Plebe(0.5);              // явное преобразование
```

В C++11 использование ключевого слова `explicit` расширено (см. главу 11) также и на функции преобразования:

```
class Plebe
{
   ...
   // Функции преобразования
   operator int() const;
   explicit operator double() const;
   ...
};
...
Plebe a, b;
int n = a;                   // автоматическое преобразование int в Plebe
double x = b;                // не разрешено
x = double(b);               // явное преобразование, разрешено
```

Инициализация членов внутри класса

Многих новичков в C++ удивляло, почему нельзя было инициализировать члены, просто предоставляя значения в определении класса. Теперь это делать можно. Синтаксис выглядит следующим образом:

```
class Session
{
   int mem1 = 10;              // инициализация внутри класса
   double mem2 {1966.54};      // инициализация внутри класса
   short mem3;
public:
   Session(){}                                                          // #1
   Session(short s) : mem3(s) {}                                        // #2
   Session(int n, double d, short s) : mem1(n), mem2(d), mem3(s) {}  // #3
   ...
};
```

Можно использовать знак = или форму инициализации с фигурными скобками, но не версию с круглыми скобками. Результат будет тем же самым, что и в случае предоставления первых двух конструкторов с элементами списка инициализации для `mem1` и `mem2`:

```
Session() : mem1(10), mem2(1966.54) {}
Session(short s) : mem1(10), mem2(1966.54), mem3(s) {}
```

Обратите внимание, что применение инициализации внутри класса позволяет избежать необходимости дублирования кода в конструкторах, тем самым уменьшая количество возможностей для ошибок и объем скучной работы для программиста.

Эти значения по умолчанию переопределяются конструктором, который содержит значения в списке инициализации членов, поэтому третий конструктор переопределяет инициализацию внутри класса.

Изменения в шаблонах и STL

В C++11 внесено множество изменений, расширяющих использование шаблонов в целом и стандартной библиотеки шаблонов (Standard Template Library – STL) – в частности. Некоторые из них относятся к самой библиотеке, а другие – к простоте использования. В этой главе уже упоминались псевдонимы шаблонов и дружественные к STL интеллектуальные указатели.

Цикл `for`, основанный на диапазоне

Цикл `for`, основанный на диапазоне (рассмотренный в главах 5 и 16), упрощает написание циклов для встроенных массивов классов, таких как `std::string` и контейнеры STL, которые имеют методы `begin()` и `end()`, идентифицирующие диапазон. Такой цикл применяет указанное действие к каждому элементу в массиве или контейнере:

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (double x : prices)
   std::cout << x << std::endl;
```

Здесь `x` принимает значение каждого элемента в `prices` по очереди. Тип `x` должен соответствовать типу элемента массива. Самый простой и безопасный путь сделать это предусматривает использование `auto` для объявления `x`; компилятор выведет тип из информации в объявлении `prices`:

```
double prices[5] = {4.99, 10.99, 6.87, 7.99, 8.49};
for (auto x : prices)
   std::cout << x << std::endl;
```

Если в намерения входит модификация элементов массива или контейнера в цикле, необходимо применять ссылочный тип:

```
std::vector<int> vi(6);
for (auto & x: vi)        // использовать ссылку, если цикл изменяет содержимое
   x = std::rand();
```

Новые контейнеры STL

К коллекции контейнеров STL стандарт C++11 добавляет следующие контейнеры: `forward_list`, `unordered_map`, `unordered_multimap`, `unordered_set` и `unordered_multiset` (см. главу 16). Контейнер `forward_list` — это односвязный список, который допускает обход только в одном направлении; он проще и экономнее в плане пространства, чем контейнер в виде двухсвязного списка. Остальные четыре контейнера поддерживают реализацию хеш-таблиц.

В C++11 также появился шаблон `array` (обсуждаемый в главах 4 и 16), для которого указываются тип элемента и фиксированное количество элементов:

```
std::array<int,360> ar;  // массив из 360 элементов int
```

Этот класс шаблона не удовлетворяет всем обычным требованиям к шаблону. Например, поскольку размер является фиксированным, нельзя использовать методы, подобные `put_back()`, которые изменяют размер контейнера. Однако шаблон `array` имеет методы `begin()` и `end()`, которые позволяют применять многие основанные на диапазоне алгоритмы STL к объектам `array`.

Новые методы STL

К списку методов STL стандарт C++11 добавляет `cbegin()` и `cend()`. Подобно `begin()` и `end()`, новые методы возвращают итераторы для первого элемента и для элемента, следующего за последним в контейнере, таким образом указывая диапазон, охватывающий все элементы. Вдобавок новые методы трактуют все элементы как `const`. Аналогично, `crbegin()` и `crend()` являются `const`-версиями `rbegin()` и `rend()`.

Более важно то, что в дополнение к традиционным конструкторам копирования и обычным операциям присваивания, контейнеры STL теперь имеют конструкторы переноса и операции присваивания с переносом. Семантика переноса рассматривается позже в этой главе.

Обновление `valarray`

Шаблон `valarray` был разработан независимо от STL, и положенное в его основу изначальное проектное решение препятствует использованию основанных на диапазоне алгоритмов STL с объектами `valarray`. В C++11 добавлены две функции, `begin()` и `end()`, которые принимают аргумент `valarray`. Они возвращают итераторы для первого элемента и для элемента, следующего за последним в объекте `valarray`, позволяя применять основанные на диапазоне алгоритмы STL (см. главу 16).

Завершение использования `export`

В C++98 было введено ключевое слово `export` в надежде создать способ разделения определений шаблонов на интерфейсные файлы, содержащие прототипы и объявления шаблонов, и файлы реализации, содержащие определения шаблонных функций

и методов. Это оказалось непрактичным, и в C++11 применение export было прекращено. Тем не менее, в стандарте ключевое слово export сохранено для возможного будущего использования.

Угловые скобки

Во избежание путаницы с операцией >>, язык C++ требует наличия пробела между угловыми скобками во вложенных объявлениях шаблонов:

```
std::vector<std::list<int> > vl;   // >> не допускается
```

В C++11 это требование отменено:

```
std::vector<std::list<int>> vl;    // >> допускается в C++11
```

Ссылка rvalue

Традиционная ссылка C++, которая теперь называется ссылкой lvalue, привязывает идентификатор к значению lvalue. Здесь lvalue — это выражение, такое как имя переменной или разыменованный указатель, представляющий данные, для которых программа может получить адрес. Первоначально lvalue было значением, которое могло встречаться в левой стороне оператора присваивания, но появление модификатора const позволило конструкциям, не допускающим присваивание, быть адресуемыми:

```
int n;
int * pt = new int;
const int b = 101;    // присваивать b не разрешено, но &b допускается
int & rn = n;         // n идентифицирует элемент данных по адресу &n
int & rt = *pt;       // *pt идентифицирует элемент данных по адресу pt
const int & rb = b;   // b идентифицирует элемент данных const по адресу &b
```

В C++11 добавлена ссылка rvalue (обсуждавшаяся в главе 8), указываемая с использованием &&, которая может привязываться к значениям rvalue — т.е. значениям, находящимся в правой стороне операции присваивания, к которым нельзя применять операцию взятия адреса. Примерами могут служить литеральные константы (за исключением строк в стиле C, которые вычисляются как адреса), выражения наподобие x + y и возвращаемые значения функций при условии, что функции не возвращают ссылки:

```
int x = 10;
int y = 23;
int && r1 = 13;
int && r2 = x + y;
double && r3 = std::sqrt(2.0);
```

Обратите внимание, что r2 в действительности привязывается к значению, которое получается в результате вычисления x + y к этому моменту. То есть r2 привязывается к значению 23, и последующие изменения x или y на r2 влияния не оказывают. Интересно отметить, что привязка rvalue к ссылке rvalue дает в итоге значение, хранящееся в ячейке, адрес которой можно получить. Это значит, что хотя нельзя применить операцию & к 13, ее можно применить к r1. Такая привязка данных к определенному адресу и делает возможным доступ к данным через ссылки rvalue.

В листинге 18.1 представлен короткий пример, иллюстрирующий некоторые аспекты ссылок rvalue.

Листинг 18.1. rvref.cpp

```
// rvref.cpp -- простое использование ссылок rvalue
#include <iostream>
inline double f(double tf) {return 5.0*(tf-32.0)/9.0;};
int main()
{
   using namespace std;
   double tc = 21.5;
   double && rd1 = 7.07;
   double && rd2 = 1.8 * tc + 32.0;
   double && rd3 = f(rd2);

   // Вывод значений и адресов tc, rd1, rd2 и rd3
   cout << " tc value and address: " << tc <<", " << &tc << endl;
   cout << "rd1 value and address: " << rd1 <<", " << &rd1 << endl;
   cout << "rd2 value and address: " << rd2 <<", " << &rd2 << endl;
   cout << "rd3 value and address: " << rd3 <<", " << &rd3 << endl;
   cin.get();
   return 0;
}
```

Ниже показан вывод программы из листинга 18.1:

```
 tc value and address: 21.5, 002FF744
rd1 value and address: 7.07, 002FF728
rd2 value and address: 70.7, 002FF70C
rd3 value and address: 21.5, 002FF6F0
```

Одной из главных причин появления ссылок rvalue является реализация семантики переноса, которая рассматривается далее в главе.

Семантика переноса и ссылка rvalue

Теперь давайте перейдем к теме, которая пока еще не обсуждалась. C++11 делает возможной методику под названием *семантика переноса* (move semantics). Это порождает ряд вопросов: что собой представляет семантика переноса, за счет чего она становится возможной в C++11 и для чего она вообще нужна? Начнем с последнего вопроса.

Необходимость в семантике переноса

Рассмотрим работу процесса копирования до C++11. Предположим, что мы начинаем со следующего кода:

```
vector<string> vstr;
// Построение вектора из 20 000 строк, каждая по 1000 символов
...
vector<string> vstr_copy1(vstr);  // сделать vstr_copy1 копией vstr
```

Классы `vector` и `string` оба используют динамическое выделение памяти, поэтому они имеют определенные конструкторы копирования, в которых применяется какая-то версия `new`. Для инициализации объекта `vstr_copy1` конструктор копирования `vector<string>` будет использовать операцию `new` для выделения памяти под 20 000 объектов `string`, а каждый объект `string`, в свою очередь, вызовет конструктор копирования `string`, который будет применять `new` для выделения памяти под 1000 символов.

Затем все 20 000 000 символов будут копироваться из памяти, управляемой vstr, в память, управляемую vstr_copy1. Это требует большой работы, но раз уж надо, так надо. Но действительно ли это надо делать? Есть ситуации, когда ответом будет: нет.

Например, предположим, что имеется функция, которая возвращает объект vector<string>:

```
vector<string> allcaps(const vector<string> & vs)
{
   vector<string> temp;
   // Код для сохранения в temp версии vs в верхнем регистре
   return temp;
}
```

Далее предположим, что эта функция используется следующим образом:

```
vector<string> vstr;
// Построение вектора из 20 000 строк, каждая по 1000 символов
vector<string> vstr_copy1(vstr);                  // #1
vector<string> vstr_copy2(allcaps(vstr));         // #2
```

На первый взгляд, операторы #1 и #2 похожи; оба они применяют существующий объект для инициализации нового объекта vector<string>. Если рассмотреть этот код буквально, allcaps() создает временный объект, который управляет 20 000 000 символов, конструкторы копирования vector и string выполняют свою работу по созданию дубликатов 20 000 000 символов, после чего программа удаляет временный объект, возвращенный allcaps(). (Некоторые компиляторы могут даже копировать временный объект во временный возвращаемый объект, удалить временный объект, а затем удалить возвращаемый объект.) Основной момент состоит в том, что здесь впустую тратится множество усилий. Поскольку временный объект удаляется, не лучше ли будет, если компилятор просто передаст права владения данными вектору vstr_copy2?

Другими словами, вместо копирования 20 000 000 символов в новое расположение и затем удаление старого расположения можно просто оставить символы на месте и присоединить к ним метку vstr_copy2. Это может быть похоже на ситуацию, когда файл перемещается из одного каталога в другой: действительный файл остается на том же месте, где он хранился на жестком диске, а меняются только учетные данные. Подобный подход и называется семантикой переноса. Хоть это и звучит несколько парадоксально, но семантика переноса действительно позволяет избежать перемещения основных данных; она только должным образом корректирует учетные данные.

Для реализации семантики переноса необходимо каким-то образом сообщать компилятору, когда должна производиться действительная копия, а когда — нет. Именно здесь вступает в игру ссылка rvalue. Мы можем определить два конструктора. Первый, обычный конструктор копирования, может использовать в качестве параметра lvalue-ссылку const. Эта ссылка будет привязана к аргументам lvalue, таким как vstr в операторе #1. Второй, конструктор переноса, может применять ссылку rvalue, которая будет привязана к аргументам rvalue, таким как возвращаемое значение allcaps(vstr) в операторе #2. Конструктор копирования может создавать обычную детальную копию, в то время как конструктор переноса может просто корректировать учетные данные. В процессе передачи прав владения новому объекту конструктор переноса может изменять свой аргумент, из чего следует, что параметр типа ссылки rvalue не должен быть const.

Пример семантики переноса

Давайте рассмотрим пример, который поможет прояснить работу семантики переноса и ссылок rvalue. В листинге 18.2 определяется и используется класс `Useless`, инкорпорирующий динамическое выделение памяти, обычный конструктор копирования и конструктор переноса, в котором применяется семантика переноса и ссылка rvalue. Для иллюстрации работы конструкторы и деструктор, как обычно, выдают сообщения, а класс использует переменную состояния, позволяющую отслеживать количество объектов. Кроме того, некоторые важные методы, такие как операция присваивания, опущены.

Листинг 18.2. `useless.cpp`

```
// useless.cpp -- класс с семантикой переноса
#include <iostream>
using namespace std;

// Интерфейс
class Useless
{
private:
    int n;                          // количество элементов
    char * pc;                      // указатель на данные
    static int ct;                  // количество объектов
    void ShowObject() const;
public:
    Useless();
    explicit Useless(int k);
    Useless(int k, char ch);
    Useless(const Useless & f);     // обычный конструктор копирования
    Useless(Useless && f);          // конструктор переноса
    ~Useless();
    Useless operator+(const Useless & f) const;

    // В версиях копирования и переноса необходима operator=()
    void ShowData() const;
};

// Реализация
int Useless::ct = 0;

Useless::Useless()
{
    ++ct;
    n = 0;
    pc = nullptr;

    // Вызов конструктора по умолчанию; вывод количества объектов
    cout << "default constructor called; number of objects: " << ct << endl;
    ShowObject();
}

Useless::Useless(int k) : n(k)
{
    ++ct;

    // Вызов конструктора int; вывод количества объектов
    cout << "int constructor called; number of objects: " << ct << endl;
    pc = new char[n];
    ShowObject();
}
```

```
Useless::Useless(int k, char ch) : n(k)
{
   ++ct;
   // Вызов конструктора int, char; вывод количества объектов
   cout << "int, char constructor called; number of objects: " << ct
        << endl;
   pc = new char[n];
   for (int i = 0; i < n; i++)
      pc[i] = ch;
   ShowObject();
}

Useless::Useless(const Useless & f): n(f.n)
{
   ++ct;
   // Вызов конструктора копирования; вывод количества объектов
   cout << "copy const called; number of objects: " << ct << endl;
   pc = new char[n];
   for (int i = 0; i < n; i++)
      pc[i] = f.pc[i];
   ShowObject();
}

Useless::Useless(Useless && f): n(f.n)
{
   ++ct;
   // Вызов конструктора переноса; вывод количества объектов
   cout << "move constructor called; number of objects: " << ct << endl;
   pc = f.pc;                        // заимствовать адрес
   f.pc = nullptr;                   // установить старый объект в нулевой указатель
   f.n = 0;
   ShowObject();
}

Useless::~Useless()
{
   // Вызов деструктора; вывод количества объектов
   cout << "destructor called; objects left: " << --ct << endl;
   cout << "deleted object:\n";
   ShowObject();
   delete [] pc;
}

Useless Useless::operator+(const Useless & f)const
{
   cout << "Entering operator+()\n";                 // начало operator+()
   Useless temp = Useless(n + f.n);
   for (int i = 0; i < n; i++)
      temp.pc[i] = pc[i];
   for (int i = n; i < temp.n; i++)
      temp.pc[i] = f.pc[i - n];
   cout << "temp object:\n";                         // временный объект
   cout << "Leaving operator+()\n";                  // конец operator+()
   return temp;
}

void Useless::ShowObject() const
{
   cout << "Number of elements: " << n;              // количество элементов
   cout << " Data address: " << (void *) pc << endl;  // адрес данных
}
```

```
void Useless::ShowData() const
{
   if (n == 0)
      cout << "(object empty)";
   else
      for (int i = 0; i < n; i++)
         cout << pc[i];
   cout << endl;
}
// Приложение
int main()
{
   {
      Useless one(10, 'x');
      Useless two = one;              // вызов конструктора копирования
      Useless three(20, 'o');
      Useless four (one + three);     // вызов operator+(), конструктора переноса
      cout << "object one: ";
      one.ShowData();
      cout << "object two: ";
      two.ShowData();
      cout << "object three: ";
      three.ShowData();
      cout << "object four: ";
      four.ShowData();
   }
}
```

Ключевыми определениями являются два конструктора копирования и переноса. Ниже приведен код конструктора копирования без операторов вывода:

```
Useless::Useless(const Useless & f): n(f.n)
{
   ++ct;
   pc = new char[n];
   for (int i = 0; i < n; i++)
      pc[i] = f.pc[i];
}
```

Этот конструктор делает обычную детальную копию и вызывается в следующем операторе:

```
Useless two = one;  // вызов конструктора копирования
```

Ссылка `f` ссылается на объект lvalue по имени `one`.

Далее приведен код конструктора переноса без операторов вывода:

```
Useless::Useless(Useless && f): n(f.n)
{
   ++ct;
   pc = f.pc;        // заимствовать адрес
   f.pc = nullptr;   // установить старый объект в нулевой указатель
   f.n = 0;
}
```

Он получает право владения существующими данными, устанавливая указатель `pc` на эти данные. В этот момент `pc` и `f.pc` указывают на одни и те же данные. Это может привести к проблемам при вызове деструкторов, поскольку операцию `delete []`

нельзя применять дважды для одного и того же адреса. Во избежание этой проблемы исходный указатель в конструкторе устанавливается в nullptr, т.к. применение delete [] к нулевому указателю не является ошибкой. Такое изъятие прав владения иногда называют *заимствованием*. В коде также устанавливается в 0 счетчик элементов исходного объекта. Это не обязательно, но делает вывод в примере более согласованным. Обратите внимание, что изменения объекта f требуют отсутствия const в объявлении параметра.

Именно этот конструктор используется в следующем операторе:

```
Useless four (one + three);  // вызов конструктора переноса
```

Выражение one + three вызывает Useless::operator+(), и ссылка rvalue по имени f привязывается к временному объекту rvalue, возвращаемому методом.

Ниже показан вывод этой программы после ее компиляции в Microsoft Visual C++ 2010:

```
int, char constructor called; number of objects: 1
Number of elements: 10 Data address: 006F4B68
copy const called; number of objects: 2
Number of elements: 10 Data address: 006F4BB0
int, char constructor called; number of objects: 3
Number of elements: 20 Data address: 006F4BF8
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 006F4C48
temp object:
Leaving operator+()
move constructor called; number of objects: 5
Number of elements: 30 Data address: 006F4C48
destructor called; objects left: 4
deleted object:
Number of elements: 0 Data address: 00000000
object one: xxxxxxxxxx
object two: xxxxxxxxxx
object three: oooooooooooooooooo
object four: xxxxxxxxxxoooooooooooooooooooo
destructor called; objects left: 3
deleted object:
Number of elements: 30 Data address: 006F4C48
destructor called; objects left: 2
deleted object:
Number of elements: 20 Data address: 006F4BF8
destructor called; objects left: 1
deleted object:
Number of elements: 10 Data address: 006F4BB0
destructor called; objects left: 0
deleted object:
Number of elements: 10 Data address: 006F4B68
```

Обратите внимание, что объект two является отдельной копией объекта one: оба отображают один и тот же вывод, но адреса данных (006F4B68 и 006F4BB0) отличаются. С другой стороны, адрес данных (006F4C48) объекта, созданного в методе Useless::operator+() — тот же, что и адрес данных, сохраненный в объекте four, который был создан конструктором переноса. Кроме того, в выводе видно, что после создания объекта four был вызван деструктор для временного объекта. Можно ска-

зать, что это временный объект, который был удален из-за того, что размер и адрес данных являются нулевыми.

Компиляция этой программы (с заменой `nullptr` на `0`) с помощью g++ 4.5.0 с флагом `-std=c++11` приводит к интересному другому выводу:

```
int, char constructor called; number of objects: 1
Number of elements: 10 Data address: 0xa50338
copy const called; number of objects: 2
Number of elements: 10 Data address: 0xa50348
int, char constructor called; number of objects: 3
Number of elements: 20 Data address: 0xa50358
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 0xa50370
temp object:
Leaving operator+()
object one: xxxxxxxxxx
object two: xxxxxxxxxx
object three: oooooooooooooooooo
object four: xxxxxxxxxxoooooooooooooooooo
destructor called; objects left: 3
deleted object:
Number of elements: 30 Data address: 0xa50370
destructor called; objects left: 2
deleted object:
Number of elements: 20 Data address: 0xa50358
destructor called; objects left: 1
deleted object:
Number of elements: 10 Data address: 0xa50348
destructor called; objects left: 0
deleted object:
Number of elements: 10 Data address: 0xa50338
```

Обратите внимание, что конструктор переноса не вызывался и было создано только четыре объекта. При создании объекта `four` компилятор не вызывал ни одного из определенных в классе конструкторов; вместо этого он вывел, что объект `four` должен быть получателем всей работы, проделанной `operator+()`, и назначил имя `four` объекту, созданному в `operator+()`. В целом компиляторы способны проводить собственную оптимизацию, если результат является таким же, как и в случае прохода через все шаги. Даже если удалить код конструктора переноса и скомпилировать программу с помощью g++, ее поведение не изменится.

Исследование конструктора переноса

Хотя использование ссылки rvalue делает возможной семантику переноса, она не включает ее автоматически. Включение предполагает выполнение двух шагов. Первый шаг состоит в том, что ссылка rvalue позволяет компилятору идентифицировать, когда семантику переноса может использоваться:

```
Useless two = one;          // соответствует Useless::Useless(const Useless &)
Useless four (one + three); // соответствует Useless::Useless(Useless &&)
```

Объект `one` является lvalue, поэтому он соответствует ссылке lvalue, а выражение `one + three` является rvalue, так что оно соответствует ссылке rvalue. Следовательно, ссылка rvalue направляет инициализацию объекта `four` на конструктор переноса.

Второй шаг включения семантики переноса заключается в написании кода конструктора переноса для обеспечения нужного поведения.

Короче говоря, наличие одного конструктора со ссылкой lvalue и еще одного со ссылкой rvalue разделяет инициализации на две группы. Объекты, инициализируемые с помощью объекта lvalue, используют конструктор копирования, а объекты, инициализируемые с помощью объекта rvalue, применяют конструктор переноса. Поведение этих конструкторов может быть разным.

Возникает вопрос: а как обстояли дела до того, как ссылки rvalue стали частью языка? Если нет никаких конструкторов переноса, и компилятор не устранит при оптимизации необходимость в использовании конструктора копирования, то что конкретно произойдет? В C++98 следующий оператор вызовет конструктор копирования:

```
Useless four (one + three);
```

Но ссылка lvalue не может быть привязана к rvalue. Что же случится? Как было показано в главе 8, ссылочный параметр `const` будет привязан к временной переменной или объекту, если фактический аргумент представляет собой rvalue:

```
int twice(const & rx) {return 2 * rx;}
...
int main()
{
    int m = 6;
    // Далее rx ссылается на m
    int n = twice(m);
    // Далее rx ссылается на временную переменную, инициализированную значением 21
    int k = twice(21);
    ...
```

Таким образом, в случае `Useless` формальный параметр `f` будет инициализирован временным объектом, который сам инициализирован возвращаемым значением `operator+()`. Ниже показана выдержка из результатов выполнения программы из листинга 18.2, скомпилированной старым компилятором, в которой опущен конструктор переноса:

```
...
Entering operator+()
int constructor called; number of objects: 4
Number of elements: 30 Data address: 01C337C4
temp object:
Leaving operator+()
copy const called; number of objects: 5
Number of elements: 30 Data address: 01C337E8
destructor called; objects left: 4
deleted object:
Number of elements: 30 Data address: 01C337C4
copy const called; number of objects: 5
Number of elements: 30 Data address: 01C337C4
destructor called; objects left: 4
deleted object:
Number of elements: 30 Data address: 01C337E8
...
```

Первым делом, внутри метода `Useless::operator+()` конструктор создает `temp`, выделяя память для 30 элементов по адресу `01C337C4`. После этого конструктор копирования создает временную копию, на которую будет ссылаться `f`, копируя инфор-

мацию в расположение 01C337E8. Затем объект temp, который находится по адресу 01C337C4, удаляется. Далее создает новый объект four, повторно используя ранее освобожденную память по адресу 01C337C4. Наконец, временный объект аргумента, использующий расположение 01C337E8, удаляется. В конечном итоге созданы три завершенных объекта, два из которых уничтожены. Семантика переноса предназначена для избавления именно от такой дополнительной работы.

Как показывает пример с g++, оптимизирующий компилятор может самостоятельно удалить дополнительное копирование, однако, применение ссылки rvalue позволяет программистам принудительно применять семантику переноса там, где это необходимо.

Присваивание

Те же самые соглашения, которые делают семантику переноса подходящей для конструкторов, применимы и к присваиванию. Ниже показан пример кода операций присваивания с копированием и переносом для класса Useless:

```
Useless & Useless::operator=(const Useless & f) // присваивание с копированием
{
   if (this == &f)
      return *this;
   delete [] pc;
   n = f.n;
   pc = new char[n];
   for (int i = 0; i < n; i++)
      pc[i] = f.pc[i];
   return *this;
}
Useless & Useless::operator=(Useless && f)     // присваивание с переносом
{
   if (this == &f)
      return *this;
   delete [] pc;
   n = f.n;
   pc = f.pc;
   f.n = 0;
   f.pc = nullptr;
   return *this;
}
```

Операция присваивания с копированием следует обычному шаблону, описанному в главе 12. Операция присваивания с переносом удаляет первоначальные данные в целевом объекте и заимствует их у исходного объекта. Важно то, что на данные указывает только один указатель, поэтому в методе этот указатель устанавливается в nullptr.

Как и в конструкторе переноса, параметр операции присваивания с переносом не является ссылкой const, поскольку метод изменяет исходный объект.

Принудительное применение переноса

Конструкторы переноса и операции присваивания с переносом работают со значениями rvalue. А что, если их необходимо использовать со значениями lvalue? Например, программа могла бы анализировать массив некоторых объектов, выбирать один объект для последующей работы и отбрасывать массив. Было бы удобно применять конструктор переноса или операцию присваивания с переносом для предохранения выбранного объекта.

Однако предположим, что вы пытаетесь сделать следующее:

```
Useless choices[10];
Useless best;
int pick;
... // выбор одного объекта, установка pick в его индекс
best = choices[pick];
```

Объект `choices[pick]` представляет собой lvalue, поэтому оператор присваивания будет использовать операцию присваивания с копированием, а не операцию присваивания с переносом. Но если сделать так, чтобы `choices[pick]` выглядело как rvalue, то применялась бы операция присваивания с переносом. Этого можно достигнуть за счет использования операции `static_cast<>` для приведения объекта к типу `Useless &&`. В C++11 для этого предлагается более простой способ – применение функции `std::move()`, которая объявлена в заголовочном файле `utility`.

В листинге 18.3 приведен пример. Здесь к классу `Useless` добавляются многословные версии операций присваивания, а из ранее многословных конструкторов и деструктора удалены операторы вывода на экран.

Листинг 18.3. `stdmove.cpp`

```
// stdmove.cpp -- использование std::move()
#include <iostream>
#include <utility>
// Интерфейс
class Useless
{
private:
   int n;                                  // количество элементов
   char * pc;                              // указатель на данные
   static int ct;                          // количество объектов
   void ShowObject() const;
public:
   Useless();
   explicit Useless(int k);
   Useless(int k, char ch);
   Useless(const Useless & f);             // обычный конструктор копирования
   Useless(Useless && f);                  // конструктор переноса
   ~Useless();
   Useless operator+(const Useless & f) const;
   Useless & operator=(const Useless & f);  // операция присваивания с копированием
   Useless & operator=(Useless && f);       // операция присваивания с переносом
   void ShowData() const;
};
// Реализация
int Useless::ct = 0;
Useless::Useless()
{
   ++ct;
   n = 0;
   pc = nullptr;
}
Useless::Useless(int k) : n(k)
{
   ++ct;
   pc = new char[n];
}
```

```
Useless::Useless(int k, char ch) : n(k)
{
   ++ct;
   pc = new char[n];
   for (int i = 0; i < n; i++)
      pc[i] = ch;
}
Useless::Useless(const Useless & f): n(f.n)
{
   ++ct;
   pc = new char[n];
   for (int i = 0; i < n; i++)
      pc[i] = f.pc[i];
}
Useless::Useless(Useless && f): n(f.n)
{
   ++ct;
   pc = f.pc;          // заимствовать адрес
   f.pc = nullptr;     // установить старый объект в нулевой указатель
   f.n = 0;
}
Useless::~Useless()
{
   delete [] pc;
}
Useless & Useless::operator=(const Useless & f) // операция присваивания с копированием
{
   std::cout << "copy assignment operator called:\n";
              // вызов операции присваивания с копированием
   if (this == &f)
      return *this;
   delete [] pc;
   n = f.n;
   pc = new char[n];
   for (int i = 0; i < n; i++)
      pc[i] = f.pc[i];
   return *this;
}
Useless & Useless::operator=(Useless && f)     // операция присваивания с переносом
{
   std::cout << "move assignment operator called:\n";
              // вызов операции присваивания с переносом
   if (this == &f)
      return *this;
   delete [] pc;
   n = f.n;
   pc = f.pc;
   f.n = 0;
   f.pc = nullptr;
   return *this;
}
Useless Useless::operator+(const Useless & f) const
{
   Useless temp = Useless(n + f.n);
   for (int i = 0; i < n; i++)
      temp.pc[i] = pc[i];
```

```
    for (int i = n; i < temp.n; i++)
        temp.pc[i] = f.pc[i - n];
    return temp;
}
void Useless::ShowObject() const
{
    std::cout << "Number of elements: " << n;                  // количество элементов
    std::cout << " Data address: " << (void *) pc << std::endl; // адрес данных
}
void Useless::ShowData() const
{
    if (n == 0)
        std::cout << "(object empty)";  // объект пуст
    else
        for (int i = 0; i < n; i++)
            std::cout << pc[i];
    std::cout << std::endl;
}
// Приложение
int main()
{
    using std::cout;
    {
        Useless one(10, 'x');
        Useless two = one +one;      // вызов конструктора с переносом
        cout << "object one: ";
        one.ShowData();
        cout << "object two: ";
        two.ShowData();
        Useless three, four;
        cout << "three = one\n";
        three = one;                 // автоматическое присваивание с копированием
        cout << "now object three = ";
        three.ShowData();
        cout << "and object one = ";
        one.ShowData();
        cout << "four = one + two\n";
        four = one + two;            // автоматическое присваивание с переносом
        cout << "now object four = ";
        four.ShowData();
        cout << "four = move(one)\n";
        four = std::move(one);       // принудительное присваивание с переносом
        cout << "now object four = ";
        four.ShowData();
        cout << "and object one = ";
        one.ShowData();
    }
}
```

Ниже показан пример запуска программы из листинга 18.3:

```
object one: xxxxxxxxxx
object two: xxxxxxxxxxxxxxxxxxxx
three = one
copy assignment operator called:
now object three = xxxxxxxxxx
and object one = xxxxxxxxxx
```

```
four = one + two
move assignment operator called:
now object four = xxxxxxxxxxxxxxxxxxxxxxxxxxxxx
four = move(one)
move assignment operator called:
now object four = xxxxxxxxxx
and object one = (object empty)
```

Как видно в выводе, присваивание объекта `one` объекту `three` вызывает операцию присваивания с копированием, а присваивание `move(one)` объекту `four` — операцию присваивания с перемещением.

Следует понимать, что функция `std::move()` не обязательно порождает операцию переноса. Предположим, например, что `Chunk` — это класс с закрытыми данными, и есть такой код:

```
Chunk one;
...
Chunk two;
two = std::move(one);  // семантика переноса?
```

Выражение `std::move(one)` является rvalue, поэтому оператор присваивания будет вызывать операцию присваивания с переносом для `Chunk`, при условии, что она была определена. Но если в классе `Chunk` операция присваивания с переносом не определена, компилятор будет использовать операцию присваивания с копированием. В случае если последняя тоже не определена, присваивание не допускается вообще.

Основная польза ссылок rvalue для большинства программистов связана отнюдь не с возможностью написания кода, в котором они используются. Напротив, их польза заключается в возможности пользоваться библиотечным кодом, в котором применяются ссылки rvalue для реализации семантики переноса. Например, классы STL теперь имеют конструкторы копирования, конструкторы переноса, операции присваивания с копированием и операции присваивания с переносом.

Новые возможности классов

В C++11 к классам были добавлены многие новые возможности в дополнение к тем, что уже упоминались в этой главе — т.е. операциям явного преобразования и инициализации членов внутри класса.

Специальные функции-члены

К существующим четырем *специальным функциям-членам* (конструктор по умолчанию, конструктор копирования, операция присваивания с копированием и деструктор) в C++11 добавлены еще две (конструктор переноса и операция присваивания с переносом). Они являются функциями-членами, которые компилятор предоставляет автоматически с учетом различных условий.

Вспомните, что конструктор по умолчанию — это конструктор, который может быть вызван без аргументов. Компилятор предоставляет его, если в классе не определено ни одного конструктора. Эта версия конструктора по умолчанию называется *явно заданным по умолчанию* (defaulted) конструктором. Явно заданный по умолчанию конструктор по умолчанию оставляет члены встроенных типов неинициализированными, а для членов, являющихся объектами классов, вызывает конструкторы по умолчанию.

Кроме того, компилятор предоставляет явно заданный по умолчанию конструктор копирования, если он не опрсделен, но требуется в коде, а теперь и явно заданный

по умолчанию конструктор переноса, если он не определен, но требуется в коде. Для класса по имени Someclass такие два явно заданных по умолчанию конструктора имеют следующие прототипы:

```
// Явно заданный по умолчанию конструктор копирования
Someclass::Someclass(const Someclass &);

// Явно заданный по умолчанию конструктор переноса
Someclass::Someclass(Someclass &&);
```

В подобных же обстоятельствах компилятор предоставляет явно заданную по умолчанию операцию присваивания с копированием и явно заданную по умолчанию операцию присваивания с переносом, которые имеют показанные ниже прототипы:

```
// Явно заданная по умолчанию операция присваивания с копированием
Someclass & Someclass::operator(const Someclass &);

// Явно заданная по умолчанию операция присваивания с переносом
Someclass & Someclass::operator(Someclass &&);
```

Наконец, компилятор предоставит деструктор, если он не был определен для класса.

С этим описанием связаны различные исключения. Если вы не определили деструктор, конструктор копирования или операцию присваивания с копированием, компилятор не будет автоматически предоставлять конструктор переноса или операцию присваивания с переносом. Если вы не определили конструктор переноса или операцию присваивания с переносом, компилятор не будет автоматически предоставлять конструктор копирования или операцию присваивания с копированием.

Явно заданный по умолчанию конструктор переноса и явно заданная по умолчанию операция присваивания с переносом работают подобно их аналогам с копированием, выполняя почленную инициализацию и копирование для встроенных типов. Для членов, являющихся объектами классов, конструкторы и операции присваивания для этих классов применяются, как если бы параметры были значениями rvalue. Это, в свою очередь, приводит к вызову конструкторов переноса и операций присваивания с переносом, если они определены, или, в противном случае — конструкторов копирования и операций присваивания с копированием, если таковые определены.

Явно заданные по умолчанию и удаленные методы

C++11 обеспечивает больший контроль над тем, какие методы используются. Предположим, что необходимо использовать явно заданную по умолчанию функцию, которая в силу обстоятельств не создается автоматически. Например, если определен конструктор переноса, то конструктор по умолчанию, конструктор копирования и операция присваивания с копированием не предоставляются. В таком случае можно воспользоваться ключевым словом default для явного объявления заданных по умолчанию версий этих методов:

```
class Someclass
{
public:
   Someclass(Someclass &&);
   Someclass() = default;     // использование конструктора по умолчанию,
                              // сгенерированного компилятором
   Someclass(const Someclass &) = default;
   Someclass & operator=(const Someclass &) = default;
   ...
};
```

Компилятор предоставляет тот же самый конструктор, который бы он предоставил автоматически, если не был определен конструктор переноса.

Ключевое слово delete, с другой стороны, служит для предотвращения использования компилятором указанного метода. Например, чтобы предотвратить копирование объекта, можно запретить конструктор копирования и операцию присваивания с копированием:

```
class Someclass
{
public:
   Someclass() = default;      // использование конструктора по умолчанию,
                               // сгенерированного компилятором
   // Запрет конструктора копирования и операции присваивания с копированием:
   Someclass(const Someclass &) = delete;
   Someclass & operator=(const Someclass &) = delete;

   // Использование конструктора переноса и операции присваивания
   // с переносом, которые сгенерированы компилятором:
   Someclass(Someclass &&) = default;
   Someclass & operator=(Someclass &&) = default;
   Someclass & operator+(const Someclass &) const;
   ...
};
```

Как было показано в главе 12, запретить копирование можно, поместив конструктор копирования и операцию присваивания с копированием в закрытый раздел класса. Однако ключевое слово delete предлагает более удобный и понятный способ.

В чем состоит эффект от запрещения методов копирования, в то время как методы переноса остаются доступными? Вспомните, что ссылка rvalue, такая как используемая операциями переноса, привязывается только к выражениям rvalue. Отсюда вытекает следующее:

```
Someclass one;
Someclass two;
Someclass three(one);          // не разрешено, one - это lvalue
Someclass four(one + two);     // разрешено, выражение - это rvalue
```

Только шесть специальных функций-членов могут быть явно заданными по умолчанию, но ключевое слово delete можно применять к любой функции-члену. Одно из возможных использований связано с запретом некоторых преобразований. Предположим, например, что класс Someclass имеет метод с параметром типа double:

```
class Someclass
{
public:
   ...
   void redo(double);
   ...
};
```

Пусть написан следующий код:

```
Someclass sc;
sc.redo(5);
```

Значение 5 типа int расширяется до 5.0, и метод redo() будет выполнен. Теперь предположим, что определение Someclass модифицировано:

```
class Someclass
{
public:
   ...
   void redo(double);
   void redo(int) = delete;
   ...
};
```

В этом случае вызов метода `sc.redo(5)` соответствует прототипу `redo(int)`. Компилятор определит этот факт и также определит, что прототип `redo(int)` удален, и затем пометит показанный вызов как ошибку времени компиляции. Это иллюстрирует важный факт, касающийся удаленных функций. Они существуют, но их использование приводит к ошибке.

Делегирование конструкторов

Во время разработки класса с множеством конструкторов может обнаружиться, что приходится писать один и тот же код снова и снова. Другими словами, некоторые конструкторы могут требовать дублирования кода, уже присутствующего в других конструкторах. Для упрощения кода и повышения его надежности C++11 позволяет использовать один конструктор как часть определения другого конструктора. Такой прием называется *делегированием*, поскольку один конструктор временно делегирует другому конструктору ответственность за работу над создаваемым объектом. Делегирование использует разновидность синтаксиса списковой инициализации членов:

```
class Notes {
   int k;
   double x;
   std::string st;
public:
   Notes();
   Notes(int);
   Notes(int, double);
   Notes(int, double, std::string);
};
Notes::Notes(int kk, double xx, std::string stt) : k(kk),
             x(xx), st(stt) {/*какие-то действия*/}
Notes::Notes() : Notes(0, 0.01, "Oh") {/*какие-то действия*/}
Notes::Notes(int kk) : Notes(kk, 0.01, "Ah") {/*какие-то действия*/ }
Notes::Notes( int kk, double xx ) : Notes(kk, xx, "Uh") {/*какие-то действия*/ }
```

Конструктор по умолчанию, например, использует первый конструктор в списке для инициализации данных-членов, а также делает все, что требуется в теле этого конструктора. После этого он завершает работу, выполнив все, что требуется в его теле.

Наследование конструкторов

В качестве дальнейшего упрощения кодирования C++11 предоставляет производным классам механизм для наследования конструкторов от базового класса. В C++98 уже имеется синтаксис для получения доступа к функциям из пространства имен:

```
namespace Box
{
   int fn(int) { ... }
   int fn(double) { ... }
   int fn(const char *) { ... }
}
using Box::fn;
```

Это делает доступными все перегруженные функции `fn`. Тот же самый прием применяется для того, чтобы сделать неспециальные функции-члены доступными производному классу. Например, взгляните на следующий код:

```
class C1
{
   ...
public:
   ...
   int fn(int j) { ... }
   double fn(double w) { ... }
   void fn(const char * s) { ... }
};
class C2 : public C1
{
   ...
public:
   ...
   using C1::fn;
   double fn(double) { ... };
};
...
C2 c2;
int k = c2.fn(3);          // используется C1::fn(int)
double z = c2.fn(2.4);     // используется C2::fn(double)
```

Объявление `using` в `C2` делает доступными объекту `C2` три метода `fn()` из `C1`. Однако методу `fn(double)`, определенному в `C2`, отдается предпочтение перед таким же методом из `C1`.

Та же методика в C++11 применяется и к конструкторам. Все конструкторы базового класса, отличные от конструкторов по умолчанию, копирования и переноса, становятся возможными конструкторами для производного класса, но конструкторы, которые имеют сигнатуры, совпадающие с сигнатурами конструкторов производного класса, не используются:

```
class BS
{
   int q;
   double w;
public:
   BS() : q(0), w(0) {}
   BS(int k) : q(k), w(100) {}
   BS(double x) : q(-1), w(x) {}
   B0(int k, double x) : q(k), w(x) {}
   void Show() const {std::cout << q <<", " << w << '\n';}
};
class DR : public BS
{
   short j;
public:
   using BS::BS;
   DR() : j(-100) {}    // DR требуется собственный конструктор по умолчанию
   DR(double x) : BS(2*x), j(int(x)) {}
   DR(int i) : j(-2), BS(i, 0.5* i) {}
   void Show() const {std::cout << j << ", "; BS::Show();}
};
```

```
int main()
{
   DR o1;             // используется DR()
   DR o2(18.81);      // используется DR(double) вместо BS(double)
   DR o3(10, 1.8);    // используется BS(int, double)
   ...
}
```

Поскольку конструктора вроде `DR(int, double)` не существует, для `o3` используется унаследованный конструктор `BS(int, double)`. Обратите внимание, что унаследованный конструктор базового класса инициализирует только члены базового класса. Если нужно инициализировать также и члены производного класса, вместо наследования можно применять синтаксис списковой инициализации:

```
DR(int i, int k, double x) : j(i), BS(k,x) {}
```

Управление виртуальными методами: `override` и `final`

Виртуальные методы являются важным компонентом реализации иерархий полиморфных классов, в рамках которых ссылка или указатель базового класса позволяет вызывать конкретный метод, соответствующий типу объекта, на который осуществляется ссылка. С виртуальными методами связаны некоторые ловушки. Например, предположим, что в базовом классе объявлен отдельный виртуальный метод, а в производном классе решено предоставить его другую версию. Это называется *переопределением* старой версии. Однако, как обсуждалось в главе 13, в случае несовпадения сигнатуры функции старая ее версия будет скрыта, а не переопределена:

```
class Action
{
   int a;
public:
   Action(int i = 0) : a(i) {}
   int val() const {return a;};
   virtual void f(char ch) const { std::cout << val() << ch << "\n"; }
};
class Bingo : public Action
{
public:
   Bingo(int i = 0) : Action(i) {}
   virtual void f(char * ch) const { std::cout << val() << ch << "!\n"; }
};
```

Поскольку в классе `Bingo` используется `f(char * ch)` вместо `f(char ch)`, версия `f(char ch)` скрывается для объекта `Bingo`. В итоге в программе не может применяться код вроде показанного ниже:

```
Bingo b(10);
b.f('@');  // работает для объекта Action, но не работает для объекта Bingo
```

В C++11 можно использовать спецификатор `override`, чтобы отразить намерение переопределить виртуальную функцию. Его необходимо размещать после списка параметров. Если объявление не соответствует базовому методу, компилятор сообщит об этом. Таким образом, следующая версия `Bingo::f()` приведет к генерации сообщения об ошибке времени компиляции:

```
virtual void f(char * ch) const override { std::cout << val()
                                           << ch << "!\n"; }
```

Например, это сообщение об ошибке в Microsoft Visual C++ 2010 выглядит примерно так:

```
method with override specifier 'override' did not override any
base class methods
```

метод со спецификатором переопределения 'override' не переопределяет ни одного метода базового класса

Спецификатор `final` решает другую проблему. Иногда может понадобиться запретить производным классам переопределение отдельного виртуального метода. Для этого необходимо указать после списка параметров спецификатор `final`. Например, следующий код будет препятствовать классам, основанным на `Action`, переопределять функцию `f()`:

```
virtual void f(char ch) const final { std::cout << val() << ch << "\n";}
```

Спецификаторы `override` и `final` — это не совсем ключевые слова. Вместо этого они называются идентификаторами со специальным назначением. Для выяснения их специального назначения компилятор использует контекст, в котором они встречаются. В других контекстах они могут применяться как обычные идентификаторы (например, как имена переменных или перечислений).

Лямбда-функции

Поначалу понятие *лямбда-функции* (или, как их еще называют, лямбда-выражения либо просто лямбда) может вам показаться не относящимся к одному из тех дополнений C++11, которые призваны помочь начинающим программистам. Увидев, как в действительности выглядит лямбда-функция, вы сочтете, что подозрения только подтвердились — и вот пример:

```
[&count](int x){count += (x % 13 == 0);}
```

Однако они не настолько загадочны, как могут выглядеть на первый взгляд, и являются исключительно полезными, особенно с алгоритмами STL, использующими функции-предикаты.

Как работают указатели на функции, функторы и лямбда

Давайте рассмотрим пример использования трех подходов к передаче информации в алгоритм STL: указатели на функции, функторы и лямбда. (Для удобства мы будем ссылаться на эти три формы как на *функциональные объекты*, поэтому дальше громоздкая формулировка "указатель на функцию, функтор или лямбда" применяться не будет.) Предположим, что необходимо сгенерировать список случайных целых чисел и выяснить, сколько из них являются кратными 3, а сколько — кратными 13.

Генерация списка исключительно прямолинейна. Один из вариантов предполагает использование массива `vector<int>` для хранения чисел и STL-алгоритма `generate()` — для заполнения массива случайными числами:

```
#include <vector>
#include <algorithm>
#include <cmath>
...
std::vector<int> numbers(1000);
std::generate(vector.begin(), vector.end(), std::rand);
```

Функция generate() принимает диапазон, указанный первыми двумя аргументами, и устанавливает каждый элемент в значение, возвращаемое третьим аргументом, который представляет собой функциональный объект, не принимающий аргументов. В этом случае функциональный объект — это указатель на стандартную функцию rand().

С помощью алгоритма count_if() несложно подсчитать количество элементов, кратных 3. В первых двух аргументах должен быть указан диапазон, как это делалось для generate().

Третий аргумент должен быть функциональным объектом, возвращающим true или false. Затем функция count_if() подсчитывает все элементы, для которых функциональный объект возвращает true. Для нахождения элементов, кратных 3, можно использовать следующее определение функции:

```
bool f3(int x) {return x % 3 == 0;}
```

Аналогично, для нахождения элементов, кратных 13, можно применять такое определение функции:

```
bool f13(int x) {return x % 13 == 0;}
```

Имея эти определения, можно подсчитать элементы:

```
int count3 = std::count_if(numbers.begin(), numbers.end(), f3);
cout << "Count of numbers divisible by 3: " << count3 << '\n';
     // количество элементов, кратных 3
int count13 = std::count_if(numbers.begin(), numbers.end(), f13);
cout << "Count of numbers divisible by 13: " << count13 << "\n\n";
     // количество элементов, кратных 13
```

Теперь давайте рассмотрим, как решить ту же задачу с использованием функтора. Функтор, как было показано в главе 16 — это класс, который можно применять так, как если бы он был именем функции, благодаря определению operator()() в качестве метода класса. Преимущество использования функтора в рассматриваемом примере заключается в том, что один и тот же функтор подходит для решения обеих задач подсчета. Ниже показано возможное определение:

```
class f_mod
{
private:
   int dv;
public:
   f_mod(int d = 1) : dv(d) {}
   bool operator()(int x) {return x % dv == 0;}
};
```

Давайте вспомним, как это работает. С помощью конструктора можно создать объект f_mod, хранящий определенное целочисленное значение:

```
f_mod obj(3);                    // f_mod.dv устанавливается в 3
```

Этот объект может использовать метод operator() для возврата значения bool:

```
bool is_div_by_3 = obj(7);       // то же, что и obj.operator()(7)
```

Сам конструктор может передаваться в качестве аргумента в такие функции, как count_if():

```
count3 = std::count_if(numbers.begin(), numbers.end(), f_mod(3));
```

Аргумент `f_mod(3)` создает объект, хранящий значение 3, а `count_if()` применяет этот созданный объект для вызова метода `operator()()`, устанавливая параметр `x` равным некоторому элементу из `numbers`. Для подсчета количества чисел, кратных не 3, а 13, в качестве третьего аргумента должно указываться `f_mod(13)`.

И, наконец, проанализируем подход с лямбда. Его название происходит от *лямбда-вычислений* — математической системы для определения и применения функций. Эта система позволяет использовать анонимные функции — т.е. позволяет обойтись без имен функций. В контексте C++11 определение анонимной функции (лямбда) можно применять в качестве аргумента для функций, ожидающих указатель на функцию или функтор. Лямбда-выражение, соответствующее функции `f3()`, выглядит следующим образом:

```
[](int x) {return x % 3 == 0;}
```

Оно очень похоже на определение `f3()`:

```
bool f3(int x) {return x % 3 == 0;}
```

Два отличия заключаются в том, что имя функции заменяется парой квадратных скобок `[]`, а возвращаемый тип не объявляется. Вместо этого возвращаемый тип будет выведен `decltype` из возвращаемого значения, и в данном случае им будет `bool`. Если лямбда не имеет оператора `return`, выводимым типом оказывается `void`. В рассматриваемом примере лямбда будет использоваться следующим образом:

```
count3 = std::count_if(numbers.begin(), numbers.end(),
                       [](int x){return x % 3 == 0;});
```

Таким образом, полное лямбда-выражение применяется так же, как указатель или конструктор функтора.

Автоматическое выведение типа для лямбда-выражений работает только в ситуациях, когда тело содержит одиночный оператор `return`. В противном случае необходимо использовать синтаксис хвостового возвращаемого типа:

```
[](double x)->double{int y = x; return x - y;}    // возвращаемым типом
                                                   // является double
```

Все сказанное иллюстрируется в листинге 18.4.

Листинг 18.4. `lambda0.cpp`

```
// lambda0.cpp -- использование лямбда-выражений
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <ctime>
const long Size1 = 39L;
const long Size2 = 100*Size1;
const long Size3 = 100*Size2;
bool f3(int x) {return x % 3 == 0;}
bool f13(int x) {return x % 13 == 0;}

int main()
{
   using std::cout;
   std::vector<int> numbers(Size1);
   std::srand(std::time(0));
   std::generate(numbers.begin(), numbers.end(), std::rand);
```

```
    // Использование указателей на функции
    cout << "Sample size = " << Size1 << '\n';      // размер выборки
    int count3 = std::count_if(numbers.begin(), numbers.end(), f3);
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
         // количество чисел, кратных 3
    int count13 = std::count_if(numbers.begin(), numbers.end(), f13);
    cout << "Count of numbers divisible by 13: " << count13 << "\n\n";
         // количество чисел, кратных 13

    // Увеличение размера numbers
    numbers.resize(Size2);
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size2 << '\n';      // размер выборки

    // Использование функтора
    class f_mod
    {
    private:
       int dv;
    public:
       f_mod(int d = 1) : dv(d) {}
       bool operator()(int x) {return x % dv == 0;}
    };
    count3 = std::count_if(numbers.begin(), numbers.end(), f_mod(3));
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
         // количество чисел, кратных 3
    count13 = std::count_if(numbers.begin(), numbers.end(), f_mod(13));
    cout << "Count of numbers divisible by 13: " << count13 << "\n\n";
         // количество чисел, кратных 13

    // Повторное увеличение размера numbers
    numbers.resize(Size3);
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size3 << '\n';      // размер выборки

    // Использование лямбда
    count3 = std::count_if(numbers.begin(), numbers.end(),
                       [](int x){return x % 3 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
         // количество чисел, кратных 3
    count13 = std::count_if(numbers.begin(), numbers.end(),
                        [](int x){return x % 13 == 0;});
    cout << "Count of numbers divisible by 13: " << count13 << '\n';
         // количество чисел, кратных 13
    return 0;
}
```

Ниже показан вывод программы из листинга 18.4:

```
Sample size = 39
Count of numbers divisible by 3: 15
Count of numbers divisible by 13: 6
Sample size = 3900
Count of numbers divisible by 3: 1305
Count of numbers divisible by 13: 302
Sample size = 390000
Count of numbers divisible by 3: 130241
Count of numbers divisible by 13: 29860
```

Вывод показывает, что не следует полагаться на статистические данные, основанные на небольших выборках.

Более подробно о лямбда-функциях

Давайте посмотрим, для чего еще служат лямбда-функции. Мы исследуем этот вопрос в терминах четырех характеристик: близость, краткость, эффективность и возможность.

Многие программисты считают полезным располагать определения близко к месту, де они используются. В этом случае не придется просматривать многие страницы исходного кода, выясняя, скажем, что собой представляет третий аргумент в вызове функции `count_if()`. Вдобавок, когда понадобится модифицировать код, все компоненты будут под рукой. И если вы вырезаете и копируете код для использования во многих местах, все необходимые компоненты также будут под рукой. С этой точки зрения лямбда-функции идеальны, поскольку их определение производится в месте применения. Обычные функции хуже, т.к. функции не могут определяться внутри других функций, поэтому возможно, что определение будет располагаться далеко от точки использования. Функторы достаточно хороши, потому что класс, в том числе и класс функтора, может быть определен внутри функции, и это позволяет располагать определение близко к месту применения.

В плане краткости код функтора более многословный, чем код эквивалентной функции или лямбда. Функции и лямбда примерно одинаково краткие. Одним очевидным исключением является ситуация, когда лямбда-функция должна использоваться дважды:

```
count1 = std::count_if(n1.begin(), n1.end(),
                  [](int x){return x % 3 == 0;});
count2 = std::count_if(n2.begin(), n2.end(),
                  [](int x){return x % 3 == 0;});
```

Но писать код лямбда-функции дважды не понадобится. В сущности можно предусмотреть имя для анонимной лямбда-функции и затем два раза воспользоваться этим именем:

```
auto mod3 = [](int x){return x % 3 == 0;}  // mod3 — это имя для лямбда
count1 = std::count_if(n1.begin(), n1.end(), mod3);
count2 = std::count_if(n2.begin(), n2.end(), mod3);
```

Эту больше не анонимную лямбда-функцию можно даже применять как обычную функцию:

```
bool result = mod3(z);  // результат равен true, если z % 3 == 0
```

Однако в отличие от обычной функции, именованная лямбда-функция может быть определена внутри функции. Действительным типом `mod3` будет некоторый зависящий от реализации тип, который компилятор применяет для отслеживания лямбда-функций.

Относительная эффективность этих трех подходов связана с тем, что именно компилятор выберет для встраивания. В данном случае эффективности подхода с указателем на функцию препятствует тот факт, что компиляторы традиционно не встраивают функцию, адрес которой используется, поскольку концепция адреса функции означает отсутствие встраивания. Относительно функторов и лямбда-функций явное противоречие со встраиванием отсутствует.

Наконец, лямбда-функции обладают некоторыми дополнительными возможностями. В частности, лямбда-функция может обращаться по имени к любой автоматической переменной в области видимости. Используемые переменные *захватываются* путем перечисления их имен внутри квадратных скобок. Когда указано только имя,

например, [z], доступ к переменной производится по значению. Если имя предваряется символом &, как в [&count], доступ к переменной осуществляется по ссылке. Применение [&] предоставляет доступ ко всем автоматическим переменным по ссылке, а [=] — доступ ко всем автоматическим переменным по значению. Можно также смешивать и сочетать. Например, [ted, &ed] обеспечит доступ к ted по значению, а к ed — по ссылке; [&, ted] предоставит доступ к ted по значению, а ко всем остальным автоматическим переменным — по ссылке; [=, &ed] обеспечит доступ к ed по ссылке, а к остальным автоматическим переменным — по значению. В листинге 18.4 можно заменить код

```
int count13;
...
count13 = std::count_if(numbers.begin(), numbers.end(),
                    [](int x){return x % 13 == 0;});
```

следующим:

```
int count13 = 0;
std::for_each(numbers.begin(), numbers.end(),
            [&count13](int x){count13 += x % 13 == 0;});
```

Конструкция [&count13] позволяет лямбда-функции использовать count13 в своем коде. Поскольку переменная count13 захвачена по ссылке, любые изменения count13 в лямбда-функции отразятся на исходной переменной count13.

Выражение x % 13 == 0 вычисляется как true, если значение x кратно 13, и при добавлении к count13 значение true преобразуется в 1. Аналогично, значение false преобразуется в 0. Таким образом, после того, как for_each() применит лямбда-выражение ко всем элементам numbers, в count13 будет храниться количество элементов, кратных 13.

Этот прием можно использовать для подсчета количества элементов, кратных 3, и элементов, кратных 13, написав единственное лямбда-выражение:

```
int count3 = 0;
int count13 = 0;
std::for_each(numbers.begin(), numbers.end(),
            [&](int x){count3 += x % 3 == 0; count13 += x % 13 == 0;});
```

На этот раз с помощью [&] все автоматически переменные, включая count3 и count13, сделаны доступными лямбда-выражению.

Описанные приемы реализованы в листинге 18.5.

Листинг 18.5. lambda1.cpp

```
// lambda1.cpp — использование захваченных переменных
#include <iostream>
#include <vector>
#include <algorithm>
#include <cmath>
#include <ctime>
const long Size = 390000L;
int main()
{
    using std::cout;
    std::vector<int> numbers(Size);
    std::srand(std::time(0));
    std::generate(numbers.begin(), numbers.end(), std::rand);
    cout << "Sample size = " << Size << '\n';     // размер выборки
```

```
    // Использование лямбда-функций
    int count3 = std::count_if(numbers.begin(), numbers.end(),
                          [](int x){return x % 3 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
          // количество чисел, кратных 3
    int count13 = 0;
    std::for_each(numbers.begin(), numbers.end(),
                [&count13](int x){count13 += x % 13 == 0;});
    cout << "Count of numbers divisible by 13: " << count13 << '\n';
          // количество чисел, кратных 13

    // Использование одиночного лямбда-выражения
    count3 = count13 = 0;
    std::for_each(numbers.begin(), numbers.end(),
                [&](int x){count3 += x % 3 == 0; count13 += x % 13 == 0;});
    cout << "Count of numbers divisible by 3: " << count3 << '\n';
          // количество чисел, кратных 3
    cout << "Count of numbers divisible by 13: " << count13 << '\n';
          // количество чисел, кратных 13
    return 0;
}
```

Ниже показан вывод программы из листинга 18.5:

```
Sample size = 390000
Count of numbers divisible by 3: 130274
Count of numbers divisible by 13: 30009
Count of numbers divisible by 3: 130274
Count of numbers divisible by 13: 30009
```

Вывод подтверждает, что оба подхода (две отдельных лямбда-функции и одиночное лямбда-выражение) в этой программе приводят к одним и тем же результатам.

Главной причиной добавления лямбда-функций к C++ было желание сделать возможным использование подобного функции выражения в качестве аргумента функции, которая ожидает на его месте указателя на функцию или функтора. Таким образом, типичной лямбда-функцией будет проверочное или сравнивающее выражение, которое может быть записано как единственный оператор `return`. Это сохраняет лямбда-функцию короткой и простой для понимания, а также делает возможным автоматическое выведение типа возвращаемого значения. Тем не менее, вполне вероятно, что сообществом программистов на C++ будут разработаны и другие сценарии использования.

Оболочки

Язык C++ предоставляет множество *оболочек* или *адаптеров*. Это объекты, которые используются для обеспечения более унифицированного и подходящего интерфейса для других программных элементов. Например, в главе 16 описаны `bind1st` и `bind2nd`, представляющие собой адаптерные функции с двумя параметрами для работы с алгоритмами STL, которые ожидают в качестве аргумента функций с одним параметром. В C++11 появились дополнительные оболочки. Они включают шаблон `bind`, являющийся более гибкой альтернативой `bind1st` и `bind2nd`, шаблон `mem_fn`, который позволяет передавать функцию-член как обычную функцию, шаблон `reference_wrapper`, позволяющий создавать объект, который действует подобно ссылке, но может быть скопирован, и оболочку `function`, предоставляющую способ унифицированной обработки множества подобных функциям форм.

Давайте рассмотрим более подробно один пример оболочки — `function` — и проанализируем, какую проблему она решает.

Оболочка `function` и неэффективность шаблонов

Пусть имеется следующая строка кода:

```
answer = ef(q);
```

Что собой представляет `ef`? Это может быть имя функции. Это может быть указатель на функцию. Это может быть функциональный объект. Это может быть имя, назначенное лямбда-выражению. Все это примеры *вызываемых типов*. Обилие вызываемых типов может привести к неэффективности шаблонов. Чтобы увидеть это, рассмотрим простой случай.

Для начала определим некоторые шаблоны в заголовочном файле, как показано в листинге 18.6.

Листинг 18.6. `somedefs.h`

```
// somedefs.h
#include <iostream>
template <typename T, typename F>
T use_f(T v, F f)
{
    static int count = 0;
    count++;
    std::cout << " use_f count = " << count
              << ", &count = " << &count << std::endl;
    return f(v);
}

class Fp
{
private:
    double z_;
public:
    Fp(double z = 1.0) : z_(z) {}
    double operator()(double p) { return z_*p; }
};

class Fq
{
private:
    double z_;
public:
    Fq(double z = 1.0) : z_(z) {}
    double operator()(double q) { return z_+ q; }
};
```

Шаблон `use_f()` использует параметр `f` для представления вызываемого типа:

```
return f(v);
```

Далее программа в листинге 18.7 вызывает шаблонную функцию `use_f()` шесть раз.

Листинг 18.7. `callable.cpp`

```
// callable.cpp -- вызываемые типы и шаблоны
#include "somedefs.h"
```

```
#include <iostream>
double dub(double x) {return 2.0*x;}
double square(double x) {return x*x;}
int main()
{
   using std::cout;
   using std::endl;
   double y = 1.21;
   cout << "Function pointer dub:\n";          // указатель на функцию dub
   cout << " " << use_f(y, dub) << endl;
   cout << "Function pointer square:\n";       // указатель на функцию square
   cout << " " << use_f(y, square) << endl;
   cout << "Function object Fp:\n";            // функциональный объект Fp
   cout << " " << use_f(y, Fp(5.0)) << endl;
   cout << "Function object Fq:\n";            // функциональный объект Fq
   cout << " " << use_f(y, Fq(5.0)) << endl;
   cout << "Lambda expression 1:\n";           // лямбда-выражение 1
   cout << " " << use_f(y, [](double u) {return u*u;}) << endl;
   cout << "Lambda expression 2:\n";           // лямбда-выражение 2
   cout << " " << use_f(y, [](double u) {return u+u/2.0;}) << endl;
   return 0;
}
```

Для каждого вызова параметр шаблона `T` установлен в тип `double`. А что с параметр шаблона `F`? Каждый раз фактический аргумент является чем-то таким, что принимает аргумент типа `double` и возвращает значение типа `double`, поэтому может показаться, что `F` будет иметь тот же самый тип для всех шести вызовов `use_f()`, и экземпляр шаблона должен создаваться только один раз. Но, как показывает следующий пример вывода, это не так:

```
Function pointer dub:
  use_f count = 1, &count = 0x402028
  2.42
Function pointer square:
  use_f count = 2, &count = 0x402028
  1.1
Function object Fp:
  use_f count = 1, &count = 0x402020
  6.05
Function object Fq:
  use_f count = 1, &count = 0x402024
  6.21
Lambda expression 1:
  use_f count = 1, &count = 0x405020
  1.4641
Lambda expression 2:
  use_f count = 1, &count = 0x40501c
  1.815
```

Шаблонная функция `use_f()` имеет статический член `count`, и с помощью его адреса можно посмотреть, сколько экземпляров было создано. В выводе присутствуют пять разных адресов, так что должно быть пять различных экземпляров шаблона `use_f()`.

Чтобы узнать, что происходит, необходимо посмотреть, как компилятор определяет тип параметра шаблона `F`. Для начала взгляните на следующий вызов:

```
use_f(y, dub);
```

Здесь dub — это имя функции, которая принимает аргумент double и возвращает значение double. Имя функции является указателем, следовательно, параметр F получает тип double (*) (double), т.е. указатель на функцию с аргументом double и возвращаемым значением double.

Далее производится такой вызов:

```
use_f(y, square);
```

Здесь снова второй аргумент имеет тип double (*) (double), поэтому данный вызов использует тот же экземпляр use_f(), что и первый вызов.

Следующие два вызова use_f() передают объекты во втором аргументе, так что F получает, соответственно, тип Fp и Fq, и мы имеем два новых экземпляра для этих значений F. Наконец, в последних двух вызовах F устанавливается в любые типы, которые компилятор использует для лямбда-выражений.

Решение проблемы

Оболочка function позволяет переписать программу так, что будет использоваться только один экземпляр use_f() вместо пяти. Обратите внимание, что указатели на функции, функциональные объекты и лямбда-выражения в листинге 18.7 разделяют общее поведение — все они требуют аргумента типа double и возвращают значение типа double. Можно сказать, что все они имеют одну и ту же *сигнатуру вызова*, описываемую возвращаемым типом, за которым следует заключенный в пару круглых скобок список типов параметров, разделенных запятыми. Таким образом, все эти шесть примеров имеют сигнатуру вызова double(double).

Шаблон function, объявленный в заголовочном файле functional, указывает объект в терминах сигнатуры вызова, и он может применяться для помещения в оболочку указателя на функцию, функционального объекта или лямбда-выражения, которые имеют одну и ту же сигнатуру вызова. Например, следующее объявление создает функциональный объект fdci, принимающий аргументы char и int и возвращающий значение типа double:

```
std::function<double(char, int)> fdci;
```

После этого fdci можно присваивать любой указатель на функцию, функциональный объект или лямбда-выражение, которые получают аргументы с типами char и int и возвращают значение типа double.

Различные вызываемые аргументы в листинге 18.7 имеют одну и ту же сигнатуру вызова — double(double). Таким образом, чтобы исправить код в листинге 18.7 и сократить количество создаваемых экземпляров, можно с помощью function<double(double)> создать шесть оболочек для шести функций, функторов и лямбда. Затем все шесть вызовов use_f() могут быть сделаны с одним и тем же типом (function<double(double)>) для F, что даст в результате только один экземпляр. Модифицированный код представлен в листинге 18.8.

Листинг 18.8. wrapped.cpp

```
// wrapped.cpp -- использование оболочки function в качестве аргумента
#include "somedefs.h"
#include <iostream>
#include <functional>
double dub(double x) {return 2.0*x;}
double square(double x) {return x*x;}
int main()
{
```

```
    using std::cout;
    using std::endl;
    using std::function;
    double y = 1.21;
    function<double(double)> ef1 = dub;
    function<double(double)> ef2 = square;
    function<double(double)> ef3 = Fq(10.0);
    function<double(double)> ef4 = Fp(10.0);
    function<double(double)> ef5 = [](double u) {return u*u;};
    function<double(double)> ef6 = [](double u) {return u+u/2.0;};
    cout << "Function pointer dub:\n";     // указатель на функцию dub
    cout << " " << use_f(y, ef1) << endl;
    cout << "Function pointer square:\n";  // указатель на функцию square
    cout << " " << use_f(y, ef2) << endl;
    cout << "Function object Fp:\n";      // функциональный объект Fp
    cout << " " << use_f(y, ef3) << endl;
    cout << "Function object Fq:\n";      // функциональный объект Fq
    cout << " " << use_f(y, ef4) << endl;
    cout << "Lambda expression 1:\n";     // лямбда-выражение 1
    cout << " " << use_f(y, ef5) << endl;
    cout << "Lambda expression 2:\n";     // лямбда-выражение 2
    cout << " " << use_f(y,ef6) << endl;
    return 0;
}
```

Ниже показан вывод программы из листинга 18.8:

```
Function pointer dub:
  use_f count = 1, &count = 0x404020
  2.42
Function pointer sqrt:
  use_f count = 2, &count = 0x404020
  1.1
Function object Fp:
  use_f count = 3, &count = 0x404020
  11.21
Function object Fq:
  use_f count = 4, &count = 0x404020
  12.1
Lambda expression 1:
  use_f count = 5, &count = 0x404020
  1.4641
Lambda expression 2:
  use_f count = 6, &count = 0x404020
  1.815
```

Как показывает вывод, для `count` используется только один адрес, а значение `count` отражает, что шаблон `use_f()` вызывался шесть раз. Так что теперь имеется только один экземпляр, вызываемый шесть раз, и это сокращает размер исполняемого кода.

Дополнительные возможности

Давайте рассмотрим еще пару вещей, которые можно делать с использованием `function`. Во-первых, в действительности мы не объявляли шесть объектов `function<double(double)>` в листинге 18.8. Вместо этого мы передали временный объект `function<double(double)>` в качестве аргумента функции `use_f()`:

```
typedef function<double(double)> fdd;  // упрощение объявления типа
// Создание и инициализация объекта с помощью dub
cout << use_f(y, fdd(dub)) << endl;
cout << use_f(y, fdd(square)) << endl;
...
```

Во-вторых, в листинге 18.8 вторые аргументы в `use_f()` адаптированы для соответствия формальному параметру `f`. Другой подход предусматривает адаптацию типа формального параметра `f` с целью соответствия исходным аргументам. Это можно сделать за счет применения функционального объекта-оболочки для второго параметра в определении шаблона `use_f()`. Определить `use_f()` можно следующим образом:

```
#include <functional>
template <typename T>
T use_f(T v, std::function<T(T)> f)     // сигнатурой вызова f является T(T)
{
   static int count = 0;
   count++;
   std::cout << " use_f count = " << count
             << ", &count = " << &count << std::endl;
   return f(v);
}
```

Тогда вызовы функции могут выглядеть так, как показано ниже:

```
cout << " " << use_f<double>(y, dub) << endl;
...
cout << " " << use_f<double>(y, Fp(5.0)) << endl;
...
cout << " " << use_f<double>(y, [](double u) {return u*u;}) << endl;
```

Аргументы `dub`, `Fp(5.0)` и т.д. сами по себе не относятся к типу `function<double(double)>`, поэтому вызовы используют `<double>` после `use_f` для указания желаемой специализации. Таким образом, `T` устанавливается в `double`, и `std::function<T(T)>` становится `std::function<double(double)>`.

Шаблоны с переменным числом аргументов

Шаблоны с переменным числом аргументов предоставляют средство создания шаблонных функций и шаблонных классов, которые принимают переменное количество аргументов. Здесь мы рассмотрим шаблонные функции с переменным числом аргументов. Например, предположим, что необходима функция, которая будет принимать любое количество параметров любого типа, при условии, что тип может быть отображен с помощью `cout`, и отображать аргументы в виде списка с разделителями-запятыми. Пусть имеется следующий код:

```
int n = 14;
double x = 2.71828;
std::string mr = "Mr. String objects!";
show_list(n, x);
show_list(x*x, '!', 7, mr);
```

Цель состоит в том, чтобы иметь возможность определить `show_list()` таким образом, чтобы обеспечить успешное компилирование этого кода и получение показанного ниже вывода:

```
14, 2.71828
7.38905, !, 7, Mr. String objects!
```

Для создания шаблонов с переменным числом аргументов потребуется понять несколько ключевых моментов:

- пакеты параметров шаблонов;
- пакеты параметров функций;
- распаковка пакета;
- рекурсия.

Пакеты параметров шаблонов и функций

Чтобы посмотреть, как работают пакеты параметров, для начала предположим, что есть простая шаблонная функция, которая отображает список, содержащий только один элемент:

```
template<typename T>
void show_list0(T value)
{
   std::cout << value << ", ";
}
```

Это определение содержит два списка параметров. Список параметров шаблона — это `T`. Список параметров функции — это `value`. Вызов функции, подобный показанному ниже, устанавливает `T` в списке параметров шаблона в `double`, а `value` в списке параметров функции — в `2.15`:

```
show_list0(2.15);
```

В C++11 появилась мета-операция троеточия (`...`), которая позволяет объявить идентификатор для пакета параметров шаблона, в сущности являющего списком типов. Аналогично она позволяет объявить идентификатор для пакета параметров функции, который, по сути, представляет собой список значений. Синтаксис выглядит следующим образом:

```
template<typename... Args>          // Args — это пакет параметров шаблона
void show_list1(Args... args)       // args — это пакет параметров функции
{
   ...
}
```

`Args` является пакетом параметров шаблона, а `args` — пакетом параметров функции. (Как и в случае имен других параметров, для этих пакетов могут использоваться любые имена, удовлетворяющие правилам идентификаторов C++.) Различие между `Args` и `T` состоит в том, что `T` соответствует одиночному типу, тогда как `Args` — любому количеству типов, включая их отсутствие. Взгляните на следующий вызов функции:

```
show_list1('S', 80, "sweet", 4.5);
```

В этом случае пакет параметров `Args` содержит такие типы, соответствующие параметрам в вызове функции: `char`, `int`, `const char *` и `double`.

Далее, почти так же, как в

```
void show_list0(T value)
```

указывается, что `value` имеет тип `T`, строка кода

```
void show_list1(Args... args)       // args — это пакет параметров функции
```

говорит о том, что `args` относится к типу `Args`. Точнее, это означает, что пакет функции `args` содержит список значений, которые соответствуют пакету шаблона `Args`,

причем как по типам, так и по их количеству. В данном случае args содержит значения 'S', 80, "sweet" и 4.5.

Таким образом, шаблон show_list1() с переменным числом аргументов может соответствовать любому из перечисленных ниже вызовов функции:

```
show_list1();
show_list1(99);
show_list1(88.5, "cat");
show_list1(2, 4, 6, 8, "who do we", std::string("appreciate"));
```

В последнем вызове пакет параметров шаблона Args будет содержать типы int, int, int, int, const char * и std::string, а пакет параметров функции args — соответствующие им значения 2, 4, 6, 8, "who do we" и std::string("appreciate").

Распаковка пакетов

Но как функция может получить доступ к содержимому этих пакетов? Никаких средств индексирования не существует. То есть нельзя использовать что-то вроде Args[2] для обращения к третьему типу в пакете. Вместо этого пакет можно распаковать, поместив троеточие справа от имени пакета параметров функции. Например, взгляните на следующий дефектный код:

```
template<typename... Args>    // Args — это пакет параметров шаблона
void show_list1(Args... args) // args — это пакет параметров функции
{
   show_list1(args...);       // передача распакованного args в show_list1()
}
```

Что это означает и почему код дефектный? Предположим, что имеется такой вызов функции:

```
show_list1(5,'L',0.5);
```

Этот вызов упаковывает значения 5, 'L' и 0.5 в args. Внутри функции вызов

```
show_list1(args...);
```

развертывается в следующий код:

```
show_list1(5,'L',0.5);
```

Другими словами, одиночная сущность args заменяется тремя значениями, сохраненными в пакете. Таким образом, конструкция args... развертывается в список отдельных аргументов функции. К сожалению, новый вызов является таким же, как исходный, поэтому функция обратится к самой себе с теми же самыми аргументами, инициировав бесконечную и ненужную рекурсию. (В этом и заключается дефект.)

Использование рекурсии в шаблонных функциях с переменным числом аргументов

Хотя рекурсия не позволяет show_list1() быть полезной функцией, правильно использованная рекурсия позволяет получать доступ к элементам пакета. Основная идея состоит в том, чтобы распаковать пакет параметров функции, обработать первый элемент в списке, передать оставшуюся часть списка рекурсивному вызову и продолжать процесс до тех пор, пока список не окажется пустым. Как обычно при рекурсии, важно удостовериться в наличии вызова, который завершает рекурсию.

Часть трюка предусматривает изменение заголовка шаблона:

```
template<typename T, typename... Args>
void show_list3(T value, Args... args)
```

При таком определении первый аргумент `show_list3()` получает тип `T` и присваивается `value`. Остальные аргументы принимаются пакетами `Args` и `args`. Это позволяет функции сделать что-то с `value`, например, отобразить значение. После этого оставшиеся аргументы в форме `args...` могут быть переданы рекурсивному вызову `show_list3()`. Каждый рекурсивный вызов затем выводит значение и передает сократившийся список до тех пор, пока список не закончится.

В листинге 18.9 приведена реализация, которая хотя и не идеально, но иллюстрирует описанный прием.

Листинг 18.9. `variadic1.cpp`

```
// variadic1.cpp -- использование рекурсии для распаковки пакета параметров
#include <iostream>
#include <string>

// Определение для 0 параметров -- завершение вызова
void show_list3() {}

// Определение для 1 и более параметров
template<typename T, typename... Args>
void show_list3( T value, Args... args)
{
    std::cout << value << ", ";
    show_list3(args...);
}

int main()
{
    int n = 14;
    double x = 2.71828;
    std::string mr = "Mr. String objects!";
    show_list3(n, x);
    show_list3(x*x, '!', 7, mr);
    return 0;
}
```

Замечания по программе

Взгляните на следующий вызов:

```
show_list3(x*x, '!', 7, mr);
```

Первый аргумент сопоставляет `T` с `double` и `value` с `x*x`. Оставшиеся три типа (`char`, `int` и `std::string`) помещаются в пакет `Args`, а оставшиеся три значения (`'!'`, `7` и `mr`) — в пакет `args`.

Затем функция `show_list3()` использует `cout` для отображения `value` (приблизительно `7.38905`) и строки `", "`. Это обеспечивает вывод первого элемента в списке.

Далее идет следующий вызов:

```
show_list3(args...);
```

С учетом развертывания `args...` он будет выглядеть следующим образом:

```
show_list3('!', 7, mr);
```

Как упоминалось ранее, список сократился на один элемент. В данный момент T и value становятся char и '!', а оставшиеся два типа и значения упаковываются, соответственно, в Args и args. Следующий рекурсивный вызов обрабатывает эти сократившиеся пакеты. Наконец, когда пакет args пуст, вызывается версия show_list3() без аргументов и процесс завершается.

Ниже показан вывод из двух вызовов функций из листинга 18.5:

```
14, 2.71828, 7.38905, !, 7, Mr. String objects!,
```

Улучшения

Функцию show_list3() можно улучшить, внеся в нее пару модификаций. Несложно заметить, что функция отображает запятую после каждого элемента в списке, но лучше, если бы она не выводила запятую за последним элементом. Это можно сделать, добавив шаблон для только одного элемента и обеспечив несколько отличающееся его поведение по сравнению с общим шаблоном:

```
// Определение для 1 параметра
template<typename T>
void show_list3(T value)
{
   std::cout << value << '\n';
}
```

Теперь, когда пакет args сокращается до одного элемента, будет вызвана эта версия, и вместо запятой она выведет символ новой строки. Поскольку рекурсивный вызов show_list3() в ней отсутствует, она также завершает рекурсию.

Вторая область для улучшений связана с тем, что текущая версия все передает по значению. Для простых типов, которые используются в примере, это нормально, но неэффективно для крупных по размеру классов, которые могут выводиться с помощью cout. Было бы лучше применять ссылки const. Посредством шаблонов с переменным числом аргументов на распаковку можно наложить образец. Вместо вызова

```
show_list3(Args... args);
```

можно записать следующий код:

```
show_list3(const Args&... args);
```

Это приведет к тому, что к каждому параметру функции будет применен образец const&. Таким образом, вместо std::string mr финальная версия параметра будет выглядеть как const std::string& mr.

Описанные два изменения отражены в листинге 18.10.

Листинг 18.10. variadic2.cpp

```
// variadic2.cpp
#include <iostream>
#include <string>

// Определение для 0 параметров
void show_list() {}

// Определение для 1 параметра
template<typename T>
void show_list(const T& value)
{
   std::cout << value << '\n';
}
```

```
// Определение для 2 и более параметров
template<typename T, typename... Args>
void show_list(const T& value, const Args&... args)
{
    std::cout << value << ", ";
    show_list(args...);
}
int main()
{
    int n = 14;
    double x = 2.71828;
    std::string mr = "Mr. String objects!";
    show_list(n, x);
    show_list(x*x, '!', 7, mr);
    return 0;
}
```

Ниже показан вывод программы из листинга 18.10:

```
14, 2.71828
7.38905, !, 7, Mr. String objects!
```

Другие средства C++11

В C++11 появилось намного больше средств, чем можно описать в одной книге, даже несмотря на то, что многие из них на момент написания книги еще не были широко внедрены. Тем не менее, полезно уделить некоторое время беглому взгляду на природу некоторых новых средств.

Параллельное программирование

В наши дни гораздо проще улучшить производительность компьютера добавлением новых процессоров, нежели увеличением скорости единственного процессора. Компьютеры с процессорами, имеющими два, четыре и более ядра, теперь стали нормой. Такие компьютеры позволяют запускать множество потоков выполнения одновременно. Один процессор может обрабатывать загрузку видеофайла, в то время как другой — обрабатывать электронную таблицу.

Некоторые действия могут выигрывать от наличия множества потоков, а некоторые нет. Представьте себе поиск чего-либо в односвязном списке. Программа должна выполняться с начала списка и следовать по ссылкам вплоть до конца списка; здесь второй поток ничем помочь не сможет. Теперь представьте несортированный массив. Используя возможность произвольного доступа массивов, можно запустить один поток с начала массива, а второй — с середины, таким образом уменьшив время поиска почти наполовину.

Множество потоков приводит к возникновению многих проблем. Что случится, если один поток зависнет или два потока попытаются получить доступ к одним и тех же данным одновременно? В C++11 проблема параллелизма решается за счет определения модели памяти, которая поддерживает многопоточное выполнение, добавления ключевого слова `thread_local` и предоставления библиотечной поддержки. Ключевое слово `thread_local` используется для объявления переменных, имеющих статическую продолжительность хранения по отношению к конкретному потоку; т.е. они прекращают существование, когда заканчивает существование поток, в котором они определены.

Библиотечная поддержка состоит из библиотеки элементарных операций, которые указаны в заголовочном файле `atomic`, и библиотеки поддержки потоков, описанных в заголовочных файлах `thread`, `mutex`, `condition_variable` и `future`.

Библиотечные дополнения

В C++11 добавлено множество специализированных библиотек. Расширяемая библиотека случайных чисел, поддерживаемая заголовочным файлом `random`, предоставляет более широкие и развитые средства генерации случайных чисел, чем `rand()`. Например, она предлагает возможность выбора генераторов случайных чисел, а также распределения, включая равномерное (подобно `rand()`), биномиальное, нормальное и многие другие.

Заголовочный файл `chrono` поддерживает работу со временами срабатывания.

Заголовочный файл `tuple` поддерживает шаблон `tuple`. Объект `tuple` — это генерализация объекта `pair`. В то время как объект `pair` может хранить два значения, типы которых не обязательно должны быть одинаковыми, объект `tuple` может хранить произвольное количество элементов различных типов.

Библиотека рациональной арифметики времени компиляции, поддерживаемая заголовочным файлом `ratio`, позволяет извлекать точное представление любого рационального числа, числитель и знаменатель которого могут быть представлены более широким целочисленным типом. Она также предоставляет арифметические операции для таких чисел.

Одним из наиболее интересных дополнений является библиотека регулярных выражений, поддерживаемая заголовочным файлом `regex`. Регулярное выражение задает образец, который может использоваться для сопоставления с содержимым текстовой строки. Например, выражение в квадратных скобках соответствует любому одиночному символу, указанному в этих скобках. Таким образом, `[cCkK]` соответствует одиночным символам `c`, `C`, `k` или `K`, а `[cCkK]at` — словам `cat`, `Cat`, `kat` и `Kat`. Другие образцы включают `\d` для цифры, `\w` для слова, `\t` для символа табуляции и т.д. Тот факт, что обратная косая черта в C++ имеет специальное значение, как начало управляющей последовательности, требует записи образца вроде `\d\t\w\d` (т.е. цифра-табуляция-слово-цифра) в виде строкового литерала `"\\d\\t\\w\\d"`, в котором для представления `\` используется `\\`. Это одна из причин ввода понятия необработанной строки (см. главу 4); она позволяет записать тот же образец как `R"\d\t\w\t"`.

Утилиты Unix, такие как `ed`, `grep` и `awk`, используют регулярные выражения, а интерпретируемый язык Perl расширяет их возможности. Библиотека регулярных выражений C++ позволяет выбирать разновидности регулярных выражений для работы.

Низкоуровневое программирование

Понятие "низкоуровневое" в низкоуровневом программировании связано с уровнем абстракции, а не качеством программирования. Низкий уровень означает близость к битам и байтам компьютерного оборудования и машинному языку. Низкоуровневое программирование важно для написания встроенных программ, а также для увеличения эффективности некоторых операций. Для тех, кто занимается низкоуровневым программированием, в C++11 предлагается ряд средств.

Одно из изменений связано с ослаблением ограничений на то, что называют POD (Plain Old Data — простой тип данных). В C++98 в качестве POD используется скалярный тип (тип с одним значением, такой как `int` или `double`) или устаревшая структура без конструкторов, базовых классов, закрытых данных, виртуальных функций и т.п. Идея заключалась в том, что POD представляет собой данные, которые безопас-

но копировать побайтно. Эта идея сохранилась, но в C++11 удалось убрать некоторые ограничения из C++98 и по-прежнему иметь жизнеспособный POD. Это полезно для низкоуровневого программирования, т.к. определенные низкоуровневые операции вроде использования функций C для побайтного копирования или бинарного ввода-вывода требуют POD.

Другое изменение касается объединений: они сделаны более гибкими за счет того, что могут содержать члены, имеющие конструкторы и деструкторы, но ограниченные другие свойства; например, виртуальные функции не разрешены. Объединения часто применяются в ситуациях, когда важно минимизировать объем используемой памяти, и новые правила позволяют программистам обеспечить в таких случаях бо́льшую гибкость и возможности.

В C++11 решены проблемы выравнивания адресов памяти. Компьютерные системы могут ограничивать способы хранения данных в памяти. Например, одна система может требовать, чтобы значения `double` сохранялись в ячейках с четными адресами, тогда как в другой системе эти значения должны сохраняться в ячейках с адресами, кратными восьми. Операция `alignof()` (см. приложение Д) сообщает сведения о требованиях выравнивания для типа или объекта. Спецификатор `alignas` устанавливает определенный контроль над используемым выравниванием.

Механизм `constexpr` расширяет возможность компилятора вычислять во время компиляции выражения, дающие в результате константные значения. Низкоуровневый аспект этого состоит в том, чтобы позволить переменным `const` сохраняться в памяти, предназначенной только для чтения, что может быть особенно полезно при разработке встроенных приложений. (Переменные, как `const`, так и обычные, которые инициализируются во время выполнения, сохраняются в памяти с произвольным доступом.)

Смешанные средства

C++11 следует примеру C99 в том, что поддерживает расширенные целочисленные типы, зависящие от реализации. Такие типы, например, могут использоваться в системе со 128-битными целыми числами. Расширенные типы поддерживаются в заголовочном файле `stdint.h` для C и в его версии для C++ по имени `cstdint`.

C++11 предоставляет механизм *литеральной операции* для создания пользовательских литералов. Применяя этот механизм, можно, например, определять бинарные литералы, такие как `1001001b`, которые соответствующая литеральная операция преобразует в целочисленное значение.

В C++ имеется инструмент для отладки, называемый `assert`. Это макроопределение во время выполнения проверяет, равно ли заданное утверждение `true`, в случае чего отображает сообщение; если же оно равно `false`, вызывается `abort()`. Утверждение будет обычно чем-нибудь, о чем программист может думать, что оно должно быть истинным в данной точке программы. В C++11 появилось ключевое слово `static_assert`, которое позволяет проверять утверждения во время компиляции. Основная причина его ввода — упрощение отладки шаблонов, для которых создание экземпляров осуществляется во время компиляции, не во время выполнения.

C++11 обеспечивает бо́льшую поддержку метапрограммирования — процесса построения программ, которые создают или модифицируют другие программы или даже сами себя. В C++ это может быть сделано во время компиляции с использованием шаблонов.

Языковые изменения

Каким образом язык программирования растет и развивается? После того, как C++ стал использоваться достаточно широко, потребность в международном стандарте стала очевидной, и контроль над языком по существу был передан комитету по стандартам — сначала комитету ANSI, затем объединенным комитетам ISO/ANSI и, наконец, ISO/IEC JTC1/SC22/WG21(Комитет по стандартам C++). Здесь ISO (International Organization for Standardization) — Международная организация по стандартизации, IEC (International Electrotechnical Commission) — Международная электротехническая комиссия, JTC1 (Joint Technical Committee 1) — Объединенный технический комитет предыдущих двух организаций, SC22 — подкомитет JTC1 по языкам программирования, а WG21 — рабочая группа SC22, занимающаяся C++.

Комитет принимает во внимание отчеты о дефектах и предложения по изменениям и расширениям языка, и пытается достичь консенсуса. Этот процесс не отличается ни высокой скоростью, ни простотой. Некоторое представление об этой теме дает книга Страуструпа *Design and Evolution of C++* (Addison-Wesley, 1994 г.). Во всяком случае, возможно спорная и неповоротливая динамика комитета по поиску консенсуса — не лучший способ поощрения расцвета множества инноваций. Да и не в том заключается главная роль комитета по стандартам.

Однако для C++ существует другой путь изменений — прямое воздействие со стороны сообщества программистов на C++. Программисты не могут независимо изменять язык, но могут создавать полезные библиотеки. Качественно спроектированные библиотеки могут расширять применимость и универсальность языка, существенно упрощая программирование. Библиотеки строятся на основе существующих средств языка, поэтому они не требуют дополнительной поддержки со стороны компиляторов. И если они реализованы с применением шаблонов, то могут распространяться в форме текстовых или заголовочных файлов.

Одним из примеров этой разновидности изменений является библиотека STL, изначально созданная Александром Степановым, и затем сделанная доступной Hewlett-Packard. Успех STL в сообществе программистов превратил ее в кандидата для первого стандарта ANSI/ISO. В действительности проектное решение, положенное в основу этой библиотеки, повлияло и на другие аспекты нового стандарта.

Проект Boost

В последнее время библиотека Boost стала важной частью программирования на C++ и существенно повлияла на C++11. Проект Boost начался в 1998 г., когда Бимен Дауэс (Beman Dawes), тогдашний председатель рабочей группы библиотеки C++, вместе с другими членами группы разработали план по созданию новых библиотек за рамками комитета по стандартам. Базовая идея состояла в том, чтобы предоставить вебсайт, который бы действовал в качестве открытого форума, куда разработчики могли бы отправлять свободные библиотеки C++. Этот проект предоставляет рекомендации по лицензированию и приемам программирования, и он требует экспертной оценки предлагаемых библиотек. Результатом стала группа высоко оцененных и часто используемых библиотек. Проект предоставил среду, в которой сообщество программистов могло тестировать и оценивать новые идеи и получать отклики о них.

На время написания этой книги проект Boost содержал свыше 100 библиотек, доступных для загрузки вместе с документацией на сайте `www.boost.org`. Большинство из библиотек могут использоваться за счет включения соответствующих заголовочных файлов.

Проект TR1

Technical Report 1 (TR1) был проектом подгруппы комитета по стандартам C++. Он описывал набор библиотечных расширений, которые совместимы со стандартом C++98, но не требуются им. Эти расширения были кандидатами для следующей версии стандарта. Библиотека TR1 позволила сообществу C++ оценить достоинства библиотечных компонентов. Таким образом, когда комитет по стандартам включил большую часть TR1 в C++11, он имел дело с известными и проверенными библиотеками.

Библиотеки Boost внесли большой вклад в TR1. Примеры включают класс шаблона `tuple`, класс шаблона `array`, шаблоны `bind` и `function`, интеллектуальные указатели (с рядом изменений в именах и реализации), `static_assert`, библиотеку `regex` и библиотеку `random`. Кроме того, опыт сообщества Boost и пользователей TR1 привел к фактическим языковым изменениям, таким как отказ от спецификаций исключений и добавление шаблонов с переменным числом аргументов, которые позволили лучше реализовать класс шаблона `tuple` и шаблон `function`.

Использование Boost

Несмотря на то что многие библиотеки Boost стали доступны как часть стандарта C++11, есть еще много дополнительных библиотек Boost, которые имеет смысл исследовать. Например, `lexical_cast` из библиотеки Conversion предоставляет простые преобразования между числовыми и строковыми типами. Синтаксис указывается после `dynamic_cast`, где предоставляется целевой тип как параметр шаблона. В листинге 18.11 показан простой пример.

Листинг 18.11. `lexcast.cpp`

```
// lexcast.cpp -- простое преобразование из float в string
#include <iostream>
#include <string>
#include "boost/lexical_cast.hpp"
int main()
{
   using namespace std;
   cout << "Enter your weight: ";          // запрос на ввод веса
   float weight;
   cin >> weight;
   string gain = "A 10% increase raises "; // увеличение веса на 10% и вывод результата
   string wt = boost::lexical_cast<string>(weight);
   gain = gain + wt + " to ";              // operator+() для строки
   weight = 1.1 * weight;
   gain = gain + boost::lexical_cast<string>(weight) + ".";
   cout << gain << endl;
   return 0;
}
```

Результаты двух запусков программы из листинга 18.11 выглядят следующим образом:

```
Enter your weight: 150
A 10% increase raises 150 to 165.
Enter your weight: 156
A 10% increase raises 156 to 171.600006.
```

Во втором запуске демонстрируется одно из ограничений шаблона `lexical_cast`; он не обеспечивает должный контроль над форматированием чисел с плавающей точ-

кой. Для этого придется воспользоваться средствами внутреннего форматирования, которые рассматривались в главе 17.

Шаблон `lexical_cast` можно также применять для преобразования строк в числовые значения.

Очевидно, что проект Boost содержит намного больше, чем описано здесь. Например, библиотека Any позволяет сохранять (и восстанавливать) неоднородную коллекцию значений и объектов разных типов в контейнере STL, используя шаблон `Any` в качестве оболочки для различных типов. Библиотека Boost Math расширяет список математических функций далеко за пределы стандартной математической библиотеки. Библиотека Boost Filesystem упрощает написание кода, который является переносимым между платформами с разными файловыми системами. Для получения дополнительных сведений о содержимом этой библиотеки и о том, как ее добавлять к разным платформам, обращайтесь на веб-сайт Boost (`www.boost.org`). Кроме того, некоторые реализации C++, например, от Cygwin, включают библиотеку Boost.

Что дальше?

После проработки материалов этой книги у вас должно быть хорошее представление о правилах C++. Тем не менее, это только начало изучения языка. Второй этап изучения предполагает эффективное использование языка, что требует немалого времени. Наиболее удачная ситуация сложится в рабочей или учебной среде, в которой придется иметь дело с качественным кодом C++ и опытными программистами. Кроме того, зная сам язык, вы можете читать книги, посвященные более сложным темам по объектно-ориентированному программированию (ООП). Некоторые полезные ресурсы можно найти в приложении З.

Одно из обещаний ООП заключается в упрощении разработки и увеличении надежности крупных проектов. Важнейшим действием подхода ООП является создание классов, представляющих ситуацию (*предметную область*), которая должна моделироваться. Поскольку реальные задачи часто отличаются высокой сложностью, нахождение подходящего набора классов может оказаться непростым. Построить сложную систему с нуля, как правило, не удается; вместо этого лучше применять итеративный, развивающийся подход. С этой целью практики в этой области разработали множество приемов и стратегий. В частности, важно делать как можно больше итераций и этапов развития во время анализа и проектирования, а не писать и переписывать фактический код.

Двумя общепринятыми приемами являются *анализ сценариев использования* и *карты CRC*. При анализе сценариев использования команда разработчиков перечисляет общие пути, или сценарии, в которых должна использоваться окончательная система, идентифицируя элементы, действия и ответственность, которые позволят предложить возможные классы и определить их функциональность. Применение карт CRC (Class/Responsibilities/Collaborators — класс/ответственность/кооперация) — это простой способ анализа таких сценариев. Команда разработчиков создает для каждого класса индексную карту. На карте указывается имя класса, ответственность класса, такая как представленные данные и выполненные действия, и кооперация класса, т.е. перечень других классов, с которыми этот класс должен взаимодействовать. После этого производится проход через сценарий с использованием интерфейса, предоставленного картами CRC. В результате появляются предположения о создании новых классов, переносе ответственности и т.п.

В более широком масштабе существуют методы систематизации для работы с целыми проектами. Последним из них является UML (Unified Modeling Language – унифицированный язык моделирования). UML – это не язык программирования; напротив, это язык для представления анализа и проектирования программного проекта. Он был разработан Гради Бучем (Grady Booch), Джимом Рамбо (Jim Rumbaugh) и Айваром Якобсоном (Ivar Jacobson), которые были также создателями трех предшествующих средств моделирования: метода Буча (Booch Method), OMT (Object Modeling Technique – методика объектного моделирования) и OOSE (Object-Oriented Software Engineering – проектирование объектно-ориентированного программного обеспечения). Язык UML является эволюционным преемником этих трех средств, и в 2005 г. ISO/IEC был опубликован стандарт для него.

В дополнение к улучшению общего понимания C++ можно заняться изучением специфических библиотек классов. Например, Microsoft и Embarcadero предлагают обширные библиотеки классов, упрощающие программирование для среды Windows, а Apple Xcode предлагает похожие средства для продуктов Apple, включая iPhone.

Резюме

Новый стандарт C++ добавил множество средств к языку. Некоторые из них предназначены для упрощения его изучения и использования. Примерами могут служить унифицированная списковая инициализация с помощью фигурных скобок, автоматическое выведение типа посредством `auto`, инициализация членов внутри класса и цикл `for` на основе диапазона. Другие изменения расширяют и проясняют проектирование классов. К таким изменениям относятся явно заданные по умолчанию и удаленные методы, делегирование конструкторов, наследование конструкторов, а также спецификаторы `override` и `final` для уточнения виртуальных функций.

Многие дополнения помогают сделать как программы, так и само программирование более эффективным. Лямбда-выражения обладают преимуществами перед указателями на функции и функторами. Шаблон `function` может использоваться для сокращения количества создаваемых экземпляров шаблона. Ссылка rvalue делает возможной семантика переноса и позволяет реализовать конструкторы переноса и операции присваивания с переносом.

Другие изменения обеспечивают улучшенные способы для решения ряда задач. Перечисления с областью видимости предоставляют больший контроль над контекстом и лежащими в основе типами для перечислений. Шаблоны `unique_ptr` и `shared_ptr` предлагают более совершенное управление памятью, выделяемой с помощью `new`.

Механизм шаблонов расширен за счет добавления `decltype`, хвостовых возвращаемых типов, псевдонимов шаблонов и шаблонов с переменным числом аргументов.

Модифицированные правила для объединений, POD, операция `alignof()`, спецификатор `alignas` и механизм `constexpr` поддерживают низкоуровневое программирование.

Многие библиотечные дополнения, включая новые классы STL, шаблон `tuple` и библиотеку `regex`, предоставляют решения для разнообразных нужд программирования.

Новый стандарт поддерживает параллельное программирование с помощью ключевого слова `thread_local` и библиотеки `atomic`.

В общем, новый стандарт улучшает удобство использования и надежность C++ как для начинающих, так и для экспертов.

Вопросы для самоконтроля

1. Перепишите следующий код с использованием синтаксиса списковой инициализации с помощью фигурных скобок; в переписанном коде необходимо отказаться от применения массива ar:

```
class Z200
{
private:
   int j;
   char ch;
   double z;
public:
   Z200(int jv, char chv, zv) : j(jv), ch(chv), z(zv) {}
   ...
};
double x = 8.8;
std::string s = "What a bracing effect!";
int k(99);
Z200 zip(200,'Z',0.675);
std::vector<int> ai(5);
int ar[5] = {3, 9, 4, 7, 1};
for (auto pt = ai.begin(), int i = 0; pt != ai.end(); ++pt, ++i)
   *pt = ai[i];
```

2. Какие вызовы функций в следующей короткой программе являются ошибочными и почему? На что ссылается аргумент типа ссылки в допустимых вызовах?

```
#include <iostream>
using namespace std;
double up(double x) { return 2.0* x;}
void r1(const double &rx) {cout << rx << endl;}
void r2(double &rx) {cout << rx << endl;}
void r3(double &&rx) {cout << rx << endl;}
int main()
{
   double w = 10.0;
   r1(w);
   r1(w+1);
   r1(up(w));
   r2(w);
   r2(w+1);
   r2(up(w));
   r3(w);
   r3(w+1);
   r3(up(w));
   return 0;
}
```

3. а. Что отобразит следующая короткая программа и почему?

```
#include <iostream>
using namespace std;
double up(double x) { return 2.0* x;}
void r1(const double &rx) {cout << "const double & rx\n";}
void r1(double &rx) {cout << "double & rx\n";}
```

```
int main()
{
   double w = 10.0;
   rl(w);
   rl(w+1);
   rl(up(w));
   return 0;
}
```

б. Что отобразит следующая короткая программа и почему?

```
#include <iostream>
using namespace std;
double up(double x) { return 2.0* x;}
void rl(double &rx) {cout << "double & rx\n";}
void rl(double &&rx) {cout << "double && rx\n";}
int main()
{
   double w = 10.0;
   rl(w);
   rl(w+1);
   rl(up(w));
   return 0;
}
```

в. Что отобразит следующая короткая программа и почему?

```
#include <iostream>
using namespace std;
double up(double x) {return 2.0* x;}
void rl(const double &rx) {cout << "const double & rx\n";}
void rl(double &&rx) {cout << "double && rx\n";}
int main()
{
   double w = 10.0;
   rl(w);
   rl(w+1);
   rl(up(w));
   return 0;
}
```

4. Назовите специальные функции-члены и укажите, что делает их специальными?

5. Предположим, что класс `Fizzle` имеет только данные-члены, как показано ниже:

```
class Fizzle
{
private:
   double bubbles[4000];
   ...
};
```

Почему этот класс не является подходящим кандидатом для пользовательского конструктора переноса?

Какое изменение в подходе к хранению 4000 значений `double` может сделать этот класс подходящим кандидатом для функции переноса?

6. Перепишите следующую короткую программу, чтобы в ней использовалось лямбда-выражение вместо `f1()`. Не изменяйте `show2()`.

```
#include <iostream>
template<typename T>
void show2(double x, T& fp) {std::cout << x << " -> " << fp(x) << '\n';}
double f1(double x) { return 1.8*x + 32;}
int main()
{
    show2(18.0, f1);
    return 0;
}
```

7. Перепишите следующую короткую и неуклюжую программу, чтобы в ней использовалось лямбда-выражение вместо функтора `Adder`. Не изменяйте `sum()`.

```
#include <iostream>
#include <array>
const int Size = 5;
template<typename T>
void sum(std::array<double,Size> a, T& fp);
class Adder
{
    double tot;
public:
    Adder(double q = 0) : tot(q) {}
    void operator()(double w) { tot +=w; }
    double tot_v () const {return tot;};
};
int main()
{
    double total = 0.0;
    Adder ad(total);
    std::array<double, Size> temp_c = {32.1, 34.3, 37.8, 35.2, 34.7};
    sum(temp_c,ad);
    total = ad.tot_v();
    std::cout << "total: " << ad.tot_v() << '\n';
    return 0;
}
template<typename T>
void sum(std::array<double,Size> a, T& fp)
{
    for(auto pt = a.begin(); pt != a.end(); ++pt)
    {
        fp(*pt);
    }
}
```

Упражнения по программированию

1. Ниже показана часть короткой программы:

```
int main()
{
    using namespace std;
    // Список double выведен из содержимого списка
    auto q = average_list({15.4, 10.7, 9.0});
    cout << q << endl;
```

```
    // Список int выведен из содержимого списка
    cout << average_list({20, 30, 19, 17, 45, 38} ) << endl;

    // Принудительное использование списка double
    auto ad = average_list<double>({'A', 70, 65.33});
    cout << ad << endl;
    return 0;
}
```

Завершите программу, написав функцию `average_list()`. Она должна быть шаблонной функцией, с параметром типа, который используется для указания вида шаблона `initialized_list`, применяемого в качестве параметра функции, а также для указания возвращаемого типа функции.

2. Ниже показано объявление класса `Cpmv`:

```
class Cpmv
{
public:
    struct Info
    {
        std::string qcode;
        std::string zcode;
    };
private:
    Info *pi;
public:
    Cpmv();
    Cpmv(std::string q, std::string z);
    Cpmv(const Cpmv & cp);
    Cpmv(Cpmv && mv);
    ~Cpmv();
    Cpmv & operator=(const Cpmv & cp);
    Cpmv & operator=(Cpmv && mv);
    Cpmv operator+(const Cpmv & obj) const;
    void Display() const;
};
```

Функция `operator+()` должна создавать объект, члены `qcode` и `zcode` которого являются результатом конкатенации соответствующих членов операндов. Напишите код, который реализует семантику переноса для конструктора переноса и операции присваивания с переносом. Напишите программу, использующую все методы класса `Cpmv`. В целях тестирования обеспечьте выдачу сообщений в методах, чтобы можно было увидеть, когда они используются.

3. Напишите и протестируйте шаблонную функцию с переменным числом аргументов `sum_values()`, которая принимает список произвольной длины с аргументами, имеющими числовые значения (смешанных типов), и возвращает сумму в виде значения `long double`.

4. Переделайте программу в листинге 16.5 для использования лямбда-выражений. В частности, замените функцию `outint()` именованным лямбда-выражением, а два случая использования функтора — двумя анонимными лямбда-выражениями.

A

Основания систем счисления

История сохранила свидетельства того, что в древних цивилизациях испозовались многие системы представления чисел. Некоторые из них, как, например, система римских цифр, непригодны для применения в арифметических задачах. С другой стороны, система представления чисел, которую придумали древнеиндийские математики, претерпев некоторые изменения, была принята в Европе и известна как арабская система представления чисел; она используется для производства вычислений в самых разных сферах деятельности человека. Современные компьютерные системы представления чисел построены на концепции заполнителя и используют нуль, преимущества которого для записи чисел стали ясны еще древнеиндийским математикам. Однако они обобщают принципы представления чисел, используемые в других системах. Поэтому, несмотря на то, что для представления чисел мы обычно пользуемся десятичной системой, о чем будет сказано в следующем разделе, в вычислительной технике часто применяются числа, имеющие основание 8 (восьмеричная система), 16 (шестнадцатеричная система) и 2 (двоичная система).

Десятичные числа (основание 10)

Способ, который мы используем для записи чисел, основан на степени 10. Например, рассмотрим число 2468. 2 соответствует двум тысячам, 4 — четырем сотням, 6 — шести десяткам и 8 — восьми единицам:

$$2468 = 2 \times 1000 + 4 \times 100 + 6 \times 10 + 8 \times 1$$

Одну тысячу можно записать как $10 \times 10 \times 10$, или 10 в третьей степени — 10^3. Используя такое обозначение, предыдущее соотношение можно записать следующим образом:

$$2468 = 2 \times 10^3 + 4 \times 10^2 + 6 \times 10^1 + 8 \times 10^0$$

Поскольку эта форма представления чисел основана на степени основания 10, мы называем ее представлением с основанием 10, или десятичным представлением. В качестве основания можно выбрать любое другое

число. Так, для записи целых чисел в языке C++ можно применять восьмеричную (основание 8) и шестнадцатеричную (основание 16) форму. (Примечание: 10^0 равно 1, как и любое другое ненулевое число в нулевой степени.)

Восьмеричные целые числа (основание 8)

Восьмеричные числа основаны на степени 8, поэтому в восьмеричной системе для записи чисел используются цифры 0–7. Для обозначения восьмеричной системы в языке C++ служит префикс 0. Это значит, что 0177 является восьмеричным. Для получения эквивалентного значения в десятичной системе можно применить степени 8:

Восьмеричная система	Десятичная система
0177	$= 1\times8^2 + 7\times8^1 + 7\times8^0$ $= 1\times64 + 7\times8 + 7\times1$ $= 127$

Поскольку в Unix для представления значений часто используется именно восьмеричная система, эта форма записи предусмотрена также в языках C и C++.

Шестнадцатеричные числа (основание 16)

Шестнадцатеричные числа основаны на степени 16. Это означает, что 10 в шестнадцатеричной системе представляет значение 16 + 0, или 16. Чтобы представить значения от 9 до шестнадцатеричного 16, нужны дополнительные знаки. Для этого в стандартной записи шестнадцатеричной системы используются буквы от a до f. Как видно в табл. А.1, в языке C++ эти буквы могут записываться как в верхнем, так и в нижнем регистре.

Таблица А.1. Шестнадцатеричные знаки

Шестнадцатеричный знак	Десятичное значение
a или A	10
b или B	11
c или C	12
d или D	13
e или E	14
f или F	15

Для идентификации шестнадцатеричной записи в языке C++ применяется запись 0x или 0X. Поэтому 0x2B3 является шестнадцатеричным значением. Чтобы получить десятичный эквивалент числа 0x2B3, можно воспользоваться степенями 16:

Шестнадцатеричное	Десятичное
0x2B3	$= 2\times16^2 + 11\times16^1 + 3\times16^0$ $= 2\times256 + 11\times16 + 3\times1$ $= 691$

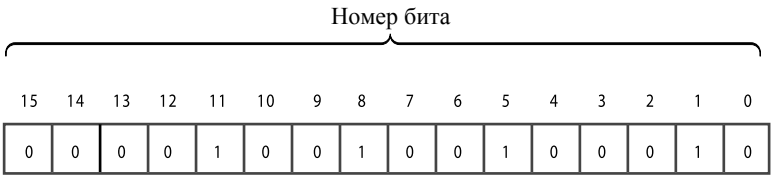
Шестнадцатеричная форма записи часто применяется в документации по оборудованию для обозначения адресов памяти и номеров портов.

Двоичные числа (основание 2)

Независимо от того, используете вы десятичную, восьмеричную или шестнадцатеричную форму для записи целого числа, в памяти компьютера оно будет храниться в двоичной форме — в виде значения с основанием 2. В двоичной записи используются всего две цифры: 0 и 1. Например, 10011011 – двоичное число. Учтите, что в C++ не предусмотрена возможность записи чисел в двоичной форме. Двоичные числа основаны на степени 2:

Двоичная запись	Десятичная запись
100110111	$= 1\times2^7 + 0\times2^6 + 0\times2^5 + 1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$ $= 128 + 0 + 0 + 16 + 8 + 0 + 2 + 1$ $= 155$

Двоичная форма записи очень удобна для памяти компьютера, в которой каждый индивидуальный элемент, называемый *битом*, может быть включен или выключен. Состояние "выключен" обозначается с помощью 0, а состояние "включен" — с помощью 1. Обычно память компьютера организована в виде элементов, называемых *байтами* или *октетами*, причем каждый байт равен 8 битам. (Как отмечалось в главе 2, байт C++ не обязательно имеет 8 бит, но в этом приложении под байтом понимается октет.) Нумерация битов в байте соответствует связанной степени с основанием 2. Поэтому самый правый бит имеет номер 0, следующий бит — 1 и т.д. Например, на рис. A.1 показано двухбайтное целое число.



$$\text{Значение} = 1 \times 2^{11} + 1 \times 2^8 + 1 \times 2^5 + 1 \times 2^1$$
$$= 2048 + 256 + 32 + 2$$
$$= 2338$$

Рис. A.1. *Двухбайтное целочисленное значение*

Двоичная и шестнадцатеричная формы записи

Шестнадцатеричная форма записи часто применяется для упрощения представления двоичных данных, например, адресов памяти или целых чисел, содержащих установки битовых флагов. Дело в том, что каждый шестнадцатеричный знак соответствует 4-битному элементу. Эти соответствия представлены в табл. A.2.

Чтобы получить из шестнадцатеричного значения двоичное, достаточно заменить каждый шестнадцатеричный знак соответствующим двоичным эквивалентом. Например, шестнадцатеричное число 0xA4 соответствует двоичному 1010 0100. Подобным образом можно выполнить обратное преобразование из двоичной формы в шестнадцатеричную, преобразуя каждый 4-битный элемент в эквивалентный шест-

надцатеричный знак. Например, шестнадцатеричным эквивалентом двоичного значения 1001 0101 является 0x95.

Таблица А.2. Шестнадцатеричные знаки и их двоичные эквиваленты

Шестнадцатеричные знаки	Двоичный эквивалент
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

Прямой и обратный порядок следования байтов.

Как это ни странно, две вычислительные платформы, в которых используется двоичное представление целых чисел, могут представлять по-разному одно и то же число. Например, процессоры Intel хранят байты с использованием схемы прямого порядка следования (Little Endian), тогда как в процессорах Motorola, мэйнфреймах IBM, процессорах SPARC и процессорах ARM реализована схема обратного порядка следования (Big Endian). (Однако последние две системы могут быть сконфигурированы на использование любой из этих схем.)

Термины *Big Endian* и *Little Endian* можно расшифровать как "Big End In" и "Little End In". Они определяют порядок расположения байтов в машинном слове (которое обычно соответствует 2-байтному элементу) памяти. В компьютере на базе процессора Intel (Little Endian) первым сохраняется младший байт. Это означает, что шестнадцатеричное значение, например `0xABCD`, будет храниться в памяти как `0xCD 0xAB`. В компьютерах на базе процессора Motorola (Big Endian) это же значение будет храниться в обратной последовательности, т.е. `0xAB 0xCD`.

Впервые эти термины были упомянуты в книге *Путешествия Гулливера* Джонатана Свифта. Свифт высмеял абсурдность многих политических диспутов на примере двух враждующих политических групп лилипутов: "тупоконечников" (Big Endians), которые утверждали, что яйцо нужно разбивать с тупого конца, и "остроконечников" (Little Endians), которые, наоборот, утверждали, что яйцо нужно разбивать с острого конца.

Специалисты по программному обеспечению должны хорошо понимать используемый порядок слов в искомой платформе. Помимо всего прочего, он влияет на интерпретацию данных, передаваемых по сети, а также на способ хранения данных в двоичных файлах. В предыдущем примере двухбайтный шаблон памяти `0xABCD` мог представлять десятичное значение 52 651 в компьютере с поддержкой схемы Little Endian, и 43 981 в компьютере с поддержкой схемы Big Endian.

Б

Зарезервированные слова C++

В языке C++ некоторые слова зарезервированы для использования как самим языком, так и его библиотеками. Зарезервированные слова нельзя применять в качестве идентификаторов в объявлениях. Зарезервированные слова делятся на три категории: ключевые слова, альтернативные лексемы и зарезервированные имена библиотеки C++.

Ключевые слова C++

Ключевые слова – это идентификаторы, формирующие словарь языка программирования. Они не могут использоваться в других целях, например, для именования переменных. В табл. Б.1 представлен список ключевых слов языка C++. Ключевые слова, выделенные полужирным, являются ключевыми словами ANSI C99. Ключевые слова, выделенные курсивом, относятся к C+11.

Таблица Б.1. Ключевые слова C++

alignas	alignof	asm	auto	bool
break	case	catch	char	char16_t
char32_t	class	const	const_cast	constexpr
continue	decltype	default	delete	do
double	dynamic_cast	else	enum	explicit
export	extern	false	float	for
friend	goto	if	inline	int
long	mutable	namespace	new	noexcept
nullptr	operator	private	protected	public
register	reinterpret_cast	return	short	signed
sizeof	static	static_assert	static_cast	struct
switch	template	this	thread_local	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	wchar_t	while		

Альтернативные лексемы

Кроме ключевых слов в C++ имеются алфавитные альтернативные представления операций, которые называются *альтернативными лексемами*. Они также являются зарезервированными. В табл. Б.2 приведен список алфавитных альтернативных лексем и соответствующих операций.

Таблица Б.2. Зарезервированные альтернативные лексемы C++ и их назначение

Лексема	Назначение
and	&&
and_eq	&=
bitand	&
bitor	|
compl	~
not	!
not_e	!=
or	||
or_eq	|=
xor	^
xor_eq	^=

Зарезервированные имена библиотеки C++

Компилятор не разрешает выбирать в качестве имен ключевые слова и альтернативные лексемы. Существует еще один класс запрещенных имен, к использованию которых компилятор относится менее строго — *зарезервированные имена*. Они представляют собой имена, зарезервированные для использования библиотекой C++. Если одно из этих имен выбрать в качестве идентификатора, то предсказать результат будет невозможно. Другими словами, это может привести к возникновению ошибки компиляции, выдаче предупреждающего сообщения, некорректному выполнению программы, а может случиться и так, что выполнение программы не будет сопровождаться ошибками.

Язык программирования C++ резервирует имена макроопределений, используемых в библиотечных заголовочных файлах. Если программа включает какой-то заголовочный файл, то применять имена макроопределений из этого файла (либо из заголовочных файлов, включенных данным файлом, и т.д.) для других целей нельзя. Например, если вы прямо или косвенно включите заголовочный файл `<climits>`, то не должны использовать `CHAR_BIT` в качестве идентификатора, поскольку это имя уже применяется в этом заголовочном файле для макроопределения.

В C++ имена, которые начинаются с двух подчеркиваний или одного подчеркивания и следующей за ним буквы верхнего регистра, зарезервированы для любого использования, а имена, которые начинаются с одного подчеркивания, зарезервированы для глобальных переменных. Следовательно, нельзя создавать имена, такие как `__gink` или `__Lynx`, в любом случае, а имена вроде `_lynx` — в глобальном пространстве имен.

В C++ имена, объявленные в библиотечных заголовочных файлах и имеющие внешнее связывание, являются зарезервированными. Применительно к функциям, это касается сигнатуры (имени и списка параметров).

Например, предположим, что имеется следующий код:

```
#include <cmath>
using namespace std;
```

В данном случае сигнатура функции tan(double) является зарезервированной. Это означает, что в вашей программе не может быть объявлена функция, которая имеет такой прототип:

```
int tan(double);      // так делать нельзя
```

Эта запись совпадает не с прототипом библиотечной функции tan(), которая возвращает тип double, но с ее сигнатурой. А вот следующий прототип использовать можно:

```
char * tan(char *);  // а так можно
```

В этой записи хоть и есть совпадение с идентификатором tan(), зато нет совпадения с сигнатурой.

Идентификаторы со специальным назначением

В сообществе C++ не склонны добавлять новые ключевые слова, поскольку в результате могут возникать конфликты с существующим кодом. Именно по этой причине комитет по стандартам, например, изменил назначение ключевого слова auto и обеспечил более одного использования для таких ключевых слов, как virtual и delete. В C++11 реализован другой способ, позволяющий избежать добавления ключевых слов — использование идентификаторов со специальным назначением. Эти идентификаторы, override и final, не являются ключевыми словами, однако они применяются для реализации языковых средств. Чтобы выяснить, как они используются — в качестве обычных идентификаторов или для реализации языковых средств — компилятор анализирует контекст:

```
class F
{
   int final;                          // #1
public:
   ...
   virtual void unfold() {...} = final; // #2
};
```

В строке #1 имя final применяется как обычный идентификатор, а в строке #2 — для обращения к языковому средству. Эти два случая использования не конфликтуют друг с другом. Кроме того, в C++ есть много идентификаторов, которые обычно присутствуют в программах, но зарезервированными не являются. К ним относятся имена заголовочных файлов, имена библиотечных функций, а также main — имя обязательной функции, с которой начинается выполнение. До тех пор, пока вы избегаете конфликтов имен, вы можете использовать эти идентификаторы для других целей, хотя никаких особых причин для этого нет. То есть ничего, кроме здравого смысла, не мешает написать код вроде показанного ниже:

```
// разрешено, но довольно неразумно
#include <iostream>
int iostream(int a);
int main ()
{
   std::cout << iostream(5) << '\n';
   return 0;
}
int iostream(int a)
{
   int main = a + 1;
   int cout = a -1;
   return main*cout;
}
```

B

Набор символов ASCII

Для хранения символов в компьютерах используются числовые коды. Наиболее распространенным кодом в США является ASCII (American Standard Code for Information Interchange – Американский стандартный код для обмена информацией). Он является подмножеством (очень малым подмножеством) кода Unicode. В языке C++ большинство символов можно представлять явным образом, заключая их в одинарные кавычки, например `'A'` для символа `A`. Кроме того, отдельный символ можно представлять посредством восьмеричного или шестнадцатеричного кода, ставя перед кодом обратную косую черту; например, коды `'\012'` и `'\0xa'` представляют один и тот же символ новой строки (LF). Такие управляющие последовательности символов могут быть частью строки, как в `"Hello,\012my dear"`.

В табл. B.1 показан набор символов ASCII и соответствующие их представления в различных системах счисления. В этой таблице символ ^, используемый в качестве префикса, обозначает нажатие клавиши <Ctrl>.

Таблица В.1 Набор символов ASCII

Десятичный код	Восьмеричный код	Шестнадцатеричный код	Двоичный код	Символ	Имя ASCII
0	0	0	00000000	^@	NUL
1	01	0x1	00000001	^A	SOH
2	02	0x2	00000010	^B	STX
3	03	0x3	00000011	^C	ETX
4	04	0x4	00000100	^D	EOT
5	05	0x5	00000101	^E	ENQ
6	06	0x6	00000110	^F	ACK
7	07	0x7	00000111	^G	BEL
8	010	0x8	00001000	^H	BS
9	011	0x9	00001001	^I, <Tab>	HT
10	012	0xa	00001010	^J	LF
11	013	0xb	00001011	^K	VT
12	014	0xc	00001100	^L	FF
13	015	0xd	00001101	^M	CR
14	016	0xe	00001110	^N	SO
15	017	0xf	00001111	^O	SI
16	020	0x10	00010000	^P	DLE
17	021	0x11	00010001	^Q	DC1
18	022	0x12	00010010	^R	DC2
19	023	0x13	00010011	^S	DC3
20	024	0x14	00010100	^T	DC4
21	025	0x15	00010101	^U	NAK
22	026	0x16	00010110	^V	SYN
23	027	0x17	00010111	^W	ETB
24	030	0x18	00011000	^X	CAN
25	031	0x19	00011001	^Y	EM
26	032	0x1a	00011010	^Z	SUB
27	033	0x1b	00011011	^[, <Esc>	ESC
28	034	0x1c	00011100	^\	FS
29	035	0x1d	00011101	^]	GS
30	036	0x1e	00011110	^^	RS
31	037	0x1f	00011111	^_	US
32	040	0x20	00100000	<Пробел>	SP
33	041	0x21	00100001	!	

Продолжение табл. B.1

Десятичный код	Восьмеричный код	Шестнадцатеричный код	Двоичный код	Символ	Имя ASCII
34	042	0x22	00100010	"	
35	043	0x23	00100011	#	
36	044	0x24	00100100	$	
37	045	0x25	00100101	%	
38	046	0x26	00100110	&	
39	047	0x27	00100111	'	
40	050	0x28	00101000	(	
41	051	0x29	00101001	)	
42	052	0x2a	00101010	*	
43	053	0x2b	00101011	+	
44	054	0x2c	00101100	'	
45	055	0x2d	00101101	-	
46	056	0x2e	00101110	.	
47	057	0x2f	00101111	/	
48	060	0x30	00110000	0	
49	061	0x31	00110001	1	
50	062	0x32	00110010	2	
51	063	0x33	00110011	3	
52	064	0x34	00110100	4	
53	065	0x35	00110101	5	
54	066	0x36	00110110	6	
55	067	0x37	00110111	7	
56	070	0x38	00111000	8	
57	071	0x39	00111001	9	
58	072	0x3a	00111010	:	
59	073	0x3b	00111011	;	
60	074	0x3c	00111100	<	
61	075	0x3d	00111101	=	
62	076	0x3e	00111110	>	
63	077	0x3f	00111111	?	
64	0100	0x40	01000000	@	
65	0101	0x41	01000001	A	
66	0102	0x42	01000010	B	
67	0103	0x43	01000011	C	

Продолжение табл. В.1

Десятичный код	Восьмеричный код	Шестнадцатеричный код	Двоичный код	Символ	Имя ASCII
68	0104	0x44	01000100	D	
69	0105	0x45	01000101	E	
70	0106	0x46	01000110	F	
71	0107	0x47	01000111	G	
72	0110	0x48	01001000	H	
73	0111	0x49	01001001	I	
74	0112	0x4a	01001010	J	
75	0113	0x4b	01001011	K	
76	0114	0x4c	01001100	L	
77	0115	0x4d	01001101	M	
78	0116	0x4e	01001110	N	
79	0117	0x4f	01001111	O	
80	0120	0x50	01010000	P	
81	0121	0x51	01010001	Q	
82	0122	0x52	01010010	R	
83	0123	0x53	01010011	S	
84	0124	0x54	01010100	T	
85	0125	0x55	01010101	U	
86	0126	0x56	01010110	V	
87	0127	0x57	01010111	W	
88	0130	0x58	01011000	X	
89	0131	0x59	01011001	Y	
90	0132	0x5a	01011010	Z	
91	0133	0x5b	01011011	[	
92	0134	0x5c	01011100	\	
93	0135	0x5d	01011101	]	
94	0136	0x5e	01011110	^	
95	0137	0x5f	01011111	_	
96	0140	0x60	01100000	'	
97	0141	0x61	01100001	a	
98	0142	0x62	01100010	b	
99	0143	0x63	01100011	c	
100	0144	0x64	01100100	d	
101	0145	0x65	01100101	e	

Окончание табл. В.1

Десятичный код	Восьмеричный код	Шестнадцатеричный код	Двоичный код	Символ	Имя ASCII
102	0146	0x66	01100110	f	
103	0147	0x67	01100111	g	
104	0150	0x68	01101000	h	
105	0151	0x69	01101001	i	
106	0152	0x6a	01101010	j	
107	0153	0x6b	01101011	k	
108	0154	0x6c	01101100	l	
109	0155	0x6d	01101101	m	
110	0156	0x6e	01101110	n	
111	0157	0x6f	01101111	o	
112	0160	0x70	01110000	p	
113	0161	0x71	01110001	q	
114	0162	0x72	01110010	r	
115	0163	0x73	01110011	s	
116	0164	0x74	01110100	t	
117	0165	0x75	01110101	u	
118	0166	0x76	01110110	v	
119	0167	0x77	01110111	w	
120	0170	0x78	01111000	x	
121	0171	0x79	01111001	y	
122	0172	0x7a	01111010	z	
123	0173	0x7b	01111011	{	
124	0174	0x7c	01111100	|	
125	0175	0x7d	01111101	}	
126	0176	0x7e	01111110	~	
127	0177	0x7f	01111111	Del	

Г

Приоритеты операций

Приоритеты операций определяет порядок, в соответствии с которым они применяются к значению. Операции в языке C++ разделены на 18 групп; все они представлены в табл. Г.1. Операции, образующие первую группу, имеют наивысший уровень приоритета выполнения; операции, относящиеся ко второй группе, занимают следующий уровень приоритета и т.д. Если две операции применены к одному и тому же операнду (некоторое значение, над которым выполняется операция), то первой будет выполнена операция, имеющая более высокий уровень приоритета. Если две операции имеют одинаковый уровень приоритета, то для определения первоочередности выполнения C++ руководствуется правилами ассоциативности. Все операции в одной группе имеют одинаковый уровень приоритета и одинаковую ассоциативность, которая может определяться слева направо или справа налево. Ассоциативность слева направо означает первоочередное выполнение самой левой операции, а ассоциативность справа налево — первоочередное выполнение самой правой операции.

Некоторые символы, например * и &, могут использоваться для более чем одной операции. В таких случаях одна форма будет (один операнд), а другая — (два операнда). Чтобы определить, какая форма имеется в виду, компилятор обращается к контексту. В табл. Г.1 отмечены группы унарных и бинарных операций, когда в каждом из случаев применяется один и тот же символ.

Далее представлено несколько примеров определения приоритетов и ассоциативности.

В следующем примере компилятору предстоит решить, какую операцию необходимо выполнить первой: сложить 5 и 3 или умножить 5 на 6:

```
3 + 5 * 6
```

Операция умножения (*) имеет более высокий приоритет, чем операция сложения (+), поэтому над операндом 5 сначала выполняется операция умножения, в результате чего выражение принимает вид 3 + 30, или 33.

Таблица Г.1. Приоритеты и ассоциативность операций C++

Операция	Ассоциативность	Назначение
Первая группа приоритетов		
`::`		Операция разрешения контекста
Вторая группа приоритетов		
(*выражение*)		Группирование
`()`	Слева направо	Вызов функции
`()`		Конструкция значения — то есть, *тип* (*выраж*)
`[]`		Индекс массива
`.`		Операция прямого членства
`->`		Операция косвенного членства
`++`		Операция инкремента, постфиксная
`--`		Операция декремента, постфиксная
`const_cast`		Специализированное приведение типа
`dynamic_cast`		Специализированное приведение типа
`reinterpret_cast`		Специализированное приведение типа
`static_cast`		Специализированное приведение типа
`typeid`		Идентификация типа
Третья группа приоритетов (все унарные)		
`!`	Справа налево	Логическое отрицание
`~`		Битовое отрицание
`+`		Унарное сложение (знак плюс)
`-`		Унарное отрицание (знак минус)
`++`		Операция инкремента, префиксная
`--`		Операция декремента, префиксная
`&`		Адрес
`*`		Разыменование (косвенное значение)
`()`		Приведение типа, т.е. (*тип*) *выражение*
`sizeof`		Размер в байтах
`alignof`		Требование выравнивания
`new`		Динамическое выделение памяти
`new []`		Динамическое выделение памяти для массива
`delete`		Динамическое освобождение памяти
`delete []`		Динамическое освобождение памяти, занимаемой массивом
`noexcept`		`false`, если операнд может сгенерировать исключение
Четвертая группа приоритетов		
`.*`	Слева направо	Разыменование члена
`->*`		Косвенное разыменование члена

Продолжение табл. Г.1

Операция	Ассоциативность	Назначение
Пятая группа приоритетов (все бинарные)		
*	Слева направо	Умножение
/		Деление
%		Модуль (остаток от целочисленного деления)
Шестая группа приоритетов (все бинарные)		
+	Слева направо	Сложение
-		Вычитание
Седьмая группа приоритетов		
<<	Слева направо	Сдвиг влево
>>		Сдвиг вправо
Восьмая группа приоритетов		
<	Слева направо	Меньше
<=		Меньше или равно
>=		Больше или равно
>		Больше
Девятая группа приоритетов		
==	Слева направо	Равно
!=		Не равно
Десятая группа приоритетов (бинарные)		
&	Слева направо	Битовое "И"
Одиннадцатая группа приоритетов		
^	Слева направо	Битовое исключающее "ИЛИ"
Двенадцатая группа приоритетов		
|	Слева направо	Битовое "ИЛИ"
Тринадцатая группа приоритетов		
&&	Слева направо	Логическое "И"
Четырнадцатая группа приоритетов		
||	Слева направо	Логическое "ИЛИ"
Пятнадцатая группа приоритетов		
:?	Справа налево	Условная операция
Шестнадцатая группа приоритетов		
=	Справа налево	Простое присваивание
*=		Умножение и присваивание
/=		Деление и присваивание
%=		Нахождение остатка и присваивание
+=		Сложение и присваивание

Окончание табл. Г.1

Операция	Ассоциативность	Назначение
`-=`		Вычитание и присваивание
`&=`		Битовое "И" и присваивание
`^=`		Битовое исключающее "ИЛИ" и присваивание
`|=`		Битовое "ИЛИ" и присваивание
`<<=`		Сдвиг влево и присваивание
`>>=`		Сдвиг вправо и присваивание
Семнадцатая группа приоритетов		
`throw`	Слева направо	Генерация исключения
Восемнадцатая группа приоритетов		
`,`	Слева направо	Комбинирование двух выражений в одно

В следующем примере компилятор должен решить, что необходимо выполнить в первую очередь: разделить `120` на `6` либо `6` умножить на `5`:

```
120 / 6 * 5
```

Обе операции — умножение и деление — имеют одинаковый уровень приоритета, а ассоциативность в данном случае определяется слева направо. Следовательно, над операндом `6` в первую очередь будет выполнена операция слева, поэтому выражение примет вид `20 * 5`, или `100`.

В следующем примере компилятор должен решить, что нужно сделать — увеличить или уменьшить `str`:

```
char * str = "Whoa";
char ch = *str++;
```

Постфиксная операция ++ имеет более высокий уровень приоритета, чем унарная операция `*`. Таким образом, операция инкремента применяется к `str`, а не к `*str`. Другими словами, после выполнения операции будет изменен не символ, на который указывает указатель, а указатель, в результате чего он будет указывать на следующий символ. Однако поскольку операция ++ является постфиксной, то инкрементирование указателя будет осуществлено после того, как переменной `ch` будет присвоено исходное значение `*str`. Таким образом, переменной `ch` присваивается символ `W`, а затем в `str` указатель перемещается на символ `h`.

Ниже приведен похожий пример:

```
char * str = "Whoa";
char ch = *++str;
```

Префиксная операция ++ и унарная операция * имеют одинаковый уровень приоритета, а ассоциативность определяется справа налево. Таким образом, и в этом случае происходит инкрементирование `str`, но не `*str`. Поскольку операция ++ имеет префиксную форму, то сначала выполняется инкрементирование `str`, а затем разыменовывается указатель. Поэтому `str` перемещается и указывает на символ `h`, после чего этот символ присваивается переменной `ch`.

Обратите внимание, что в табл. Г.1 при описании приоритета две операции, обозначаемые одним символом, делятся на и , например, унарная адресная операция и бинарная операция "И".

В приложении Б даются альтернативные представления для некоторых операций.

Д

Другие операции

Ради экономии места в главах этой книги не были рассмотрены три группы операций. К первой группе относятся битовые операции, с помощью которых можно манипулировать индивидуальными битами в значении; эти операции унаследованы из языка C. Вторая группа содержит операции разыменования членов; они были введены в C++. Третья группа включает операции, появившиеся в C++11: `alignof` и `noexcept`. Все эти операции кратко описаны в этом приложении.

Битовые операции

Битовые операции выполняются над битами целочисленных значений. Например, операция сдвига влево перемещает биты влево, а операция битового отрицания переключает каждую единицу в нуль и наоборот. В языке C++ всего насчитывается шесть битовых операций: `<<`, `>>`, `~`, `&`, `|` и `^`.

Операции сдвига

Операция сдвига влево имеет следующий синтаксис:

```
значение << сдвиг
```

Здесь *значение* — это целочисленное значение, к которому будет применена операция сдвига, а *сдвиг* — количество битов сдвига. Например, следующее выражение сдвигает все биты значения `13` на три позиции влево:

```
13 << 3
```

При этом три бита слева выходят за пределы значения и отбрасываются, а новообразованные позиции справа заполняются нулями (рис. Д.1).

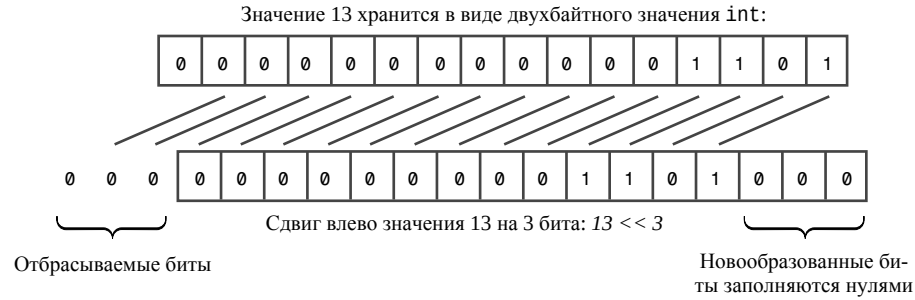


Рис. Д.1. *Операция сдвига влево*

Поскольку значение в каждой позиции бита представляет удвоенное значение бита, находящегося справа (см. приложение А), то смещение на одну позицию влево эквивалентно умножению на 2. Точно так же смещение на две позиции эквивалентно умножению на 2^2, а смещение на n позиций — умножению на 2^n. Таким образом, результатом операции `13 << 3` является 13×2^3, или 104.

Операция сдвига влево похожа на такую же операцию в языке ассемблера. Однако в языке ассемблера эта операция приводит к изменению содержимого регистра, а в языке C++ образуется новое значение без изменения существующих значений. Рассмотрим, например, следующий фрагмент кода:

```
int x = 20;
int y = x << 3;
```

Этот код не изменяет значение `x`. Выражение `x << 3` использует значение `x` для получения нового значения, подобно тому, как в выражении `x + 3` образуется новое значение без изменения содержимого `x`.

Чтобы в результате выполнения операции сдвига влево изменить значение переменной, необходимо использовать присваивание. Для этого можно применить обычное присваивание или операцию `<<=`, которая объединяет сдвиг и присваивание:

```
x = x << 4;          // обычное присваивание
y <<= 2;             // сдвиг и присваивание
```

Операция сдвига вправо (`>>`) осуществляет сдвиг битов вправо. Она имеет следующий синтаксис:

```
значение >> сдвиг
```

Здесь *значение* — это целочисленное значение, которое будет сдвинуто, а *сдвиг* — количество битов сдвига. Например, следующее выражение сдвигает все биты в значении `17` на две позиции вправо:

```
17 >> 2
```

Для беззнаковых целых чисел новообразованные позиции слева заполняются нулями, а биты, выходящие за границы значения, отбрасываются. Для целых чисел со знаком новообразованные позиции могут быть заполнены нулями или значением исходного крайнего левого бита. Выбор варианта зависит от реализации C++. (На рис. Д.2 показан пример заполнения нулями.)

Сдвиг на одну позицию вправо эквивалентен целочисленному делению на 2. В общем случае сдвиг на n позиций вправо эквивалентен целочисленному делению на 2^n.

Рис. Д.2. *Операция сдвига вправо*

В языке C++ определена также операция сдвига вправо с присваиванием, которую можно использовать для замены значения переменной значением после сдвига:

```
int q = 43;
q >>= 2;          // заменяет 43 значением 43 >> 2, или 10
```

В некоторых системах использование операций сдвига вправо и влево позволяет ускорить целочисленное умножение и деление на 2 по сравнению с операцией деления, однако, в связи с тем, что компилятор производит оптимизацию кода, эти различия будут едва заметными.

Логические битовые операции

Логические битовые операции аналогичны обычным логическим операциям, за исключением того, что они выполняются над отдельными битами значения, а не над значением в целом. Например, рассмотрим обычную операцию отрицания (`!`) и операцию битового отрицания (или нахождение дополнительного кода числа — `~`). Операция `!` преобразует значение `true` (ненулевое) в `false`, и значение `false` в `true`. Операция `~` преобразует каждый индивидуальный бит на противоположный (`1` в `0` и `0` в `1`). Например, рассмотрим значение 3, имеющее тип `unsigned char`:

```
unsigned char x = 3;
```

Выражение `!x` имеет значение `0`. Чтобы определить значение `~x`, его необходимо записать в двоичной форме: `00000011`. Затем потребуется преобразовать каждый `0` в `1` и наоборот. В результате получится значение `11111100`, которому в десятичной системе будет соответствовать `252`. (На рис. Д.3 показан пример 16-битного эквивалента.) Новое значение называется *дополнением* исходного значения.

Рис. Д.3. *Операция логического отрицания*

Битовая операция "ИЛИ" комбинирует два целочисленных значения с целью получения нового целочисленного значения. Каждый бит в новом значении устанавливается в 1, если один или другой либо оба соответствующих бита в исходных значениях установлены в 1. Если соответствующие биты установлены в 0, то результирующий бит устанавливается в 0 (рис. Д.4).

Рис. Д.4. *Битовая операция "ИЛИ"*

В табл. Д.1 описан порядок комбинирования битов при выполнении операции `|`.

Таблица Д.1. Результат выполнения операции `b1 | b2`

Значения битов	`b1 = 0`	`b1 = 1`
`b2 = 0`	0	1
`b2 = 1`	1	1

Операция `!=` комбинирует битовую операцию "ИЛИ" с присваиванием:

```
a != b;  // переменной a присваивается результат a | b
```

Битовая операция исключающего "ИЛИ" (`^`) комбинирует два целочисленных значения для получения нового целочисленного значения. Каждый бит в новом значении устанавливается в 1, если один или другой либо оба соответствующих бита в исходных значениях установлены в 1. Если оба соответствующих бита установлены в 0 или оба они установлены в 1, то результирующий бит будет установлен в 0 (рис. Д.5).

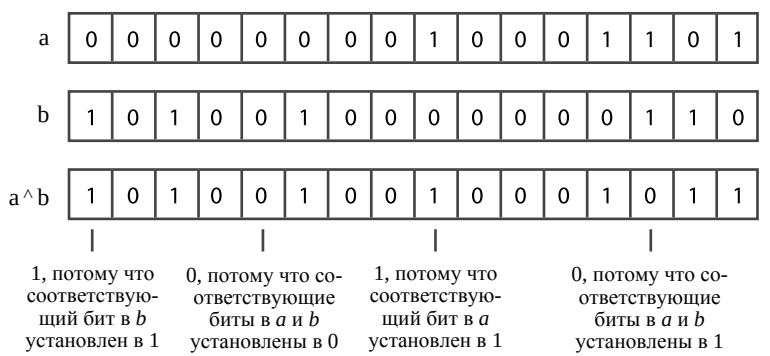


Рис. Д.5. *Битовая операция исключающего "ИЛИ"*

В табл. Д.2 описан порядок комбинирования битов при выполнении операции ^.

Таблица Д.2. Результат выполнения операции `b1 ^ b2`

Значения, присвоенные битам	`b1 = 0`	`b1 = 1`
`b2 = 0`	0	1
`b2 = 1`	1	0

Операция ^= комбинирует битовую операцию исключающего "ИЛИ" с присваиванием:

```
a ^= b;  // присваивает переменной a результат операции a ^ b
```

Битовая операция "И" (&) комбинирует два целочисленных значения для получения нового целочисленного значения. Каждый бит в новом значении устанавливается в 1, если только оба соответствующих бита в исходных значениях установлены в 1. Если хотя бы один из соответствующих битов установлен в 0, то результирующий бит будет установлен в 0 (рис. Д.6).

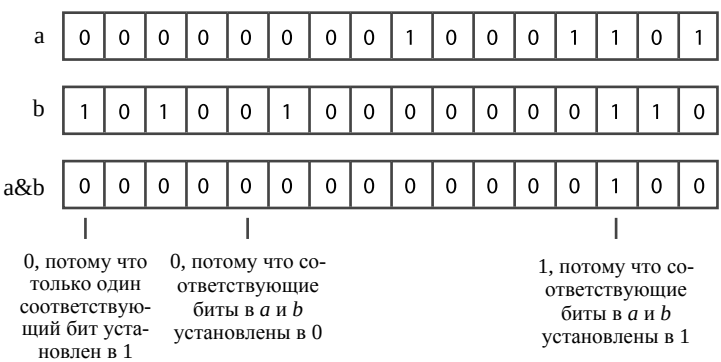


***Рис. Д.6.** Битовая операция "И"*

В табл. Д.3 описан порядок комбинирования битов при выполнении операции &.

Таблица Д.3. Результат выполнения операции `b1 & b2`

Значения, присвоенные битам	`b1 = 0`	`b1 = 1`
`b2 = 0`	0	0
`b2 = 1`	0	1

Операция &= комбинирует битовую операцию "И" с присваиванием:

```
a & b;  // присваивает переменной a результат операции a & b
```

Альтернативные представления битовых операций

В C++ доступны альтернативные представления некоторых битовых операций, показанные в табл. Д.4. Они предназначены для тех случаев, когда набор символов не позволяет использовать традиционные представления битовых операций.

Таблица Д.4. Представления битовых операций

Стандартное представление	Альтернативное представление
`&`	`bitand`
`&=`	`and_eq`
`|`	`bitor`
`|=`	`or_eq`
`~`	`compl`
`^`	`xor`
`^=`	`xor_eq`

Альтернативные представления позволяют записывать операции вроде следующих:

```
b = compl a bitand b;        // то же, что и b = ~a & b;
c = a xor b;                 // то же, что и c = a ^ c;
```

Примеры использования битовых операций

Часто при управлении аппаратными устройствами возникает необходимость во включении или выключении битов или в проверке их состояния. Эти действия можно выполнять с помощью битовых операций. Эти методы кратко рассматриваются ниже.

В следующих примерах `lottabits` представляет основное значение, а `bit` — значение, соответствующее определенному биту. Биты пронумерованы справа налево, начиная с нулевого, поэтому значение, соответствующее биту n, равно 2^n. Например, целое число, в котором установлен в 1 только третий бит, имеет значение 2^3, или 8. В общем случае каждый индивидуальный бит соответствует степени 2, как было показано в приложении А. Таким образом, мы будем использовать термин *бит* для обозначения степени 2; это будет соответствовать ситуации, когда определенный бит установлен в 1, а все остальные биты — в 0.

Включение бита

Следующие две операции включают бит в `lottabits`, который соответствует биту, представленному значением `bit`:

```
lottabits = lottabits | bit;
lottabits |= bit;
```

Каждая операция присваивает соответствующему биту единицу, вне зависимости от предыдущего значения бита. Это объясняется тем, что операция "ИЛИ" для `1` и `0` либо `1` дает `1`. Все остальные биты в `lottabits` остаются неизмененными. Это объясняется тем, что операция "ИЛИ" для `0` и `0` дает `0`, а для `0` и `1` — `1`.

Переключение бита

Следующие операции переключают бит в `lottabits`, который соответствует биту, представленному значением `bit`. Другими словами, они включают бит, если он выключен, и выключают, если он включен:

```
lottabits = lottabits ^ bit;
lottabits ^= bit;
```

Операция исключающего "ИЛИ" для 1 и 0 даст 1, включая ранее выключенный бит, а операция исключающего "ИЛИ" для 1 и 1 дает 0, выключая включенный бит. Все остальные биты в lottabits остаются неизмененными. Это объясняется тем, что исключающее "ИЛИ" для 0 и 0 дает 0, а для 0 и 1 — 1.

Выключение бита

Следующий оператор выключает бит в lottabits, который соответствует биту, представленному значением bit:

```
lottabits = lottabits & ~bit;
```

Этот оператор выключает бит независимо от его предыдущего состояния. Сначала операция ~bit дает целое число, каждый бит которого установлен в 1, *кроме* того бита, который изначально был установлен в 1; этот бит будет хранить нулевое значение. Операция "И" для 0 и любого бита дает 0, таким образом, выключая этот бит. Все остальные биты в lottabits остаются неизмененными. Это объясняется тем, что операция "И" для 1 и любого бита дает то же значение, которое хранилось в этом бите.

Далее показана более короткая запись этого же оператора:

```
lottabits &= ~bit;
```

Проверка значения бита

Предположим, что требуется проверить, равен ли 1 бит, указанный с помощью bit, в lottabits. Следующая проверка вряд ли будет работать:

```
if (lottabits == bit)          // ничего хорошего
```

Причина в том, что даже если соответствующий бит в lottabits хранит 1, то другие биты также могут иметь 1. Вышеприведенное равенство справедливо тогда, когда в 1 установлен *только* соответствующий бит. Чтобы решить эту проблему, необходимо сначала применить операцию "И" к lottabits и bit. В результате ее выполнения будет получено значение 0 во всех остальных битах, поскольку "И" для 0 и любого значения дает 0. Неизмененным останется только тот бит, который будет соответствовать значению бита, поскольку операция "И" для 1 и любого значения дает в результате это же значение. Таким образом, подходящий вариант выглядит следующим образом:

```
if (lottabits & bit == bit)    // проверка бита
```

Обычно программисты упрощают эту запись до такого вида:

```
if (lottabits & bit)           // проверка бита
```

Поскольку в bit один бит установлен в 1, а остальные биты — в 0, то результатом операции lottabits & bit будет либо 0 (что равносильно false), либо bit, что, будучи ненулевым значением, соответствует true.

Операции разыменования членов

Язык C++ позволяет определять указатели на члены класса. Эти указатели включают специальные обозначения для их объявления и разыменования. Чтобы посмотреть, что включает указатель, для начала рассмотрим простой класс:

```
class Example
{
private:
   int feet;
   int inches;
```

```
public:
   Example();
   Example(int ft);
   ~Example();
   void show_in() const;
   void show_ft() const;
   void use_ptr() const;
};
```

Рассмотрим член inches этого класса. Без определенного объекта inches представляет собой метку. Другими словами, класс определяет inches как идентификатор члена, однако вам нужен объект, прежде на самом деле будет выделена память:

```
Example ob;                    // теперь ob.inches существует
```

Таким образом, реальную ячейку памяти определяется за счет использования идентификатора inches вместе с определенным объектом. (В функции-члене можно опустить имя объекта, однако впоследствии объект будет восприниматься как тот, на который указывает указатель.)

Указатель на член для идентификатора inches можно определить следующим образом:

```
int Example::*pt = &Example::inches;
```

Этот указатель немного отличается от обычного указателя. Обычный указатель указывает на определенную ячейку памяти. А указатель pt не указывает на определенную ячейку памяти, поскольку в объявлении не идентифицирован определенный объект. Наоборот, указатель pt идентифицирует местоположение члена inches в объекте Example. Подобно идентификатору inches, идентификатор pt предназначен для использования вместе с идентификатором объекта. В сущности, выражение *pt играет роль идентификатора inches. Таким образом, идентификатор объекта можно применять для того, чтобы определить, к какому объекту производится доступ, а указатель pt — для того, чтобы определить член inches этого объекта. Например, в методе класса может присутствовать следующий код:

```
int Example::*pt = &Example::inches;
Example ob1;
Example ob2;
Example *pq = new Example;
cout << ob1.*pt << endl;       // отображает член inches объекта ob1
cout << ob2.*pt << endl;       // отображает член inches объекта ob2
cout << po->pt << endl;        // отображает член inches объекта *po
```

Здесь .* и -> представляют собой *операции разыменования членов*. Когда имеется определенный объект, например, ob1, то ob1.*pi идентифицирует член inches объекта ob1. Аналогично, pq->*pt идентифицирует член inches объекта, на который указывает pq. В предыдущем примере при изменении объекта изменялся используемый член inches. Однако можно изменить и сам указатель pt. Поскольку feet имеет тот же тип, что и inches, можно переопределить указатель pt, чтобы он указывал на член feet, а не на член inches; впоследствии ob1.*pt будет указывать на член feet объекта ob1:

```
pt = &Example::feet;           // переопределение pt
cout << ob1.*pt << endl;       // отображает член feet объекта ob1
```

По сути, комбинация *pt замещает имя члена и может использоваться для идентификации различных имен членов (такого же типа).

Указатели на члены можно применять и для идентификации функций-членов. Синтаксис этой операции несколько запутан. Вспомните, что объявление указателя на функцию void() обычного типа без аргументов выглядит следующим образом:

```
void (*pf)();                    // pf указывает на функцию
```

Объявление указателя на функцию-член необходимо для того, чтобы показать, что функция принадлежит определенному классу. Далее представлен пример объявления указателя на метод класса Example:

```
void (Example::*pf)() const;  // pf указывает на функцию-член Example
```

Этот пример показывает, что pf может использоваться точно так же, как и метод Example. Обратите внимание, что элемент Example::*pf должен быть заключен в скобки. Для этого указателя можно присвоить адрес определенной функции-члена:

```
pf = &Example::show_inches;
```

В отличие от присваивания указателя на обычную функцию здесь вы можете и должны использовать адресную операцию. Выполнив присваивание, можно будет использовать объект для вызова функции-члена:

```
Example ob3(20);
(ob3.*pf)();          // вызывает show_inches() с использованием объекта ob3
```

Всю конструкцию ob3.*pf необходимо заключить в скобки, чтобы идентифицировать выражение, которое представляет имя функции.

Поскольку show_feet() имеет ту же форму прототипа, что и show_inches(), то pf можно использовать также и для доступа к методу show_feet():

```
pf = &Example::show_feet;
(ob3.*pf)();                     // применяет show_feet() к объекту ob3
```

В определении класса, представленного в листинге Д.1, имеется метод use_ptr(), который использует указатели на члены для доступа к элементам данных и функциям-членам класса Example.

Листинг Д.1. memb_pt.cpp

```
// memb_pt.cpp -- разыменование указателей на члены классов
#include <iostream>
using namespace std;
class Example
{
private:
   int feet;
   int inches;
public:
   Example();
   Example(int ft);
   ~Example();
   void show_in() const;
   void show_ft() const;
   void use_ptr() const;
};
Example::Example()
{
   feet = 0;
   inches = 0;
}
```

```
Example::Example(int ft)
{
   feet = ft;
   inches = 12 * feet;
}
Example::~Example()
{
}
void Example::show_in() const
{
   cout << inches << " дюймов\n";
}
void Example::show_ft() const
{
   cout << feet << " футов\n";
}
void Example::use_ptr() const
{
   Example yard(3);
   int Example::*pt;
   pt = &Example::inches;
   cout << "Set pt to &Example::inches:\n";     // установка pt в &Example::inches
   cout << "this->pt: " << this->*pt << endl;
   cout << "yard.*pt: " << yard.*pt << endl;
   pt = &Example::feet;
   cout << "Set pt to &Example::feet:\n";       // установка pt в &Example::feet
   cout << "this->pt: " << this->*pt << endl;
   cout << "yard.*pt: " << yard.*pt << endl;
   void (Example::*pf)() const;
   pf = &Example::show_in;
   cout << "Set pf to &Example::show_in:\n";    // установка pt в &Example::show_in
   cout << "Using (this->*pf)(): ";             // использование (this->*pf)()
        (this->*pf)();
   cout << "Using (yard.*pf)(): ";              // использование (yard.*pf)()
        (yard.*pf)();
}
int main()
{
   Example car(15);
   Example van(20);
   Example garage;
   cout << "car.use_ptr() output:\n";           // вывод из car.use_ptr()
   car.use_ptr();
   cout << "\nvan.use_ptr() output:\n";         // вывод из van.use_ptr()
   van.use_ptr();
   return 0;
}
```

Ниже показан пример выполнения программы из листинга Д.1:

```
car.use_ptr() output:
Set pt to &Example::inches:
this->pt: 180
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 15
yard.*pt: 3
Set pf to &Example::show_in:
```

```
Using (this->*pf)(): 180 inches
Using (yard.*pf)(): 36 inches
van.use_ptr() output:
Set pt to &Example::inches:
this->pt: 240
yard.*pt: 36
Set pt to &Example::feet:
this->pt: 20
yard.*pt: 3
Set pf to &Example::show_in:
Using (this->*pf)(): 240 inches
Using (yard.*pf)(): 36 inches
```

В этом примере указателю присваиваются значения во время компиляции. В более сложном классе можно использовать указатели на элементы данных и методы, для которых точный член, связанный с указателем, определяется во время выполнения.

alignof (C++11)

Компьютерные системы могут накладывать ограничения на то, как данные хранятся в памяти. Например, одна система может требовать, чтобы значение `double` хранилось в ячейках памяти с четными адресами, тогда как в другой системе может быть необходимо, чтобы область памяти начиналась с ячейки, адрес которой кратен 8. Операция `alignof` получает тип в качестве аргумента и возвращает целое число, указывающее требуемый вид выравнивания. Требования к выравниванию могут, например, определять организацию информации внутри структуры, как показано в листинге Д.2.

Листинг Д.2. `align.cpp`

```
// align.cpp — проверка выравнивания
#include <iostream>
using namespace std;
struct things1
{
   char ch;
   int a;
   double x;
};
struct things2
{
   int a;
   double x;
   char ch;
};
int main()
{
   things1 th1;
   things2 th2;
   cout << "char alignment: " << alignof(char) << endl;       // выравнивание char
   cout << "int alignment: " << alignof(int) << endl;         // выравнивание int
   cout << "double alignment: " << alignof(double) << endl;   // выравнивание double
   cout << "things1 alignment: " << alignof(things1) << endl; // выравнивание things1
   cout << "things2 alignment: " << alignof(things2) << endl; // выравнивание things2
   cout << "things1 size: " << sizeof(things1) << endl;       // размер things1
   cout << "things2 size: " << sizeof(things2) << endl;       // размер things2
   return 0;
}
```

Ниже показан вывод в одной из систем:

```
char alignment: 1
int alignment: 4
double alignment: 8
things1 alignment: 8
things2 alignment: 8
things1 size: 16
things2 size: 24
```

Обе структуры имеют выравнивание, соответствующее 8. Одно из следствий такого выравнивания состоит в том, что размер структуры должен быть кратен 8, поэтому можно создавать массивы структур, в которых каждый элемент прилегает к следующему и также начинается с адресов, кратных 8.

Отдельные члены структур из листинга Д.2 используют всего лишь 13 бит, но требование использования количества бит, кратного восьми, означает, что каждой структуре нужно как-то заполнить лишние биты. Кроме того, внутри каждой структуры член double должен быть выровнен по адресу, кратному 8. Различное выравнивание членов структур things1 и things2 приводит к тому, что things2 нуждается в большем внутреннем заполнении, чтобы удовлетворить наложенным ограничениям.

noexcept (C++11)

Ключевое слово noexcept используется для указания, что функция не должна генерировать исключения. Его также можно применять в качестве операции, которая определяет, может ли ее операнд (выражение) потенциально сгенерировать исключение. Она возвращает false, если операнд может сгенерировать исключение, и true — в противном случае. Например, взгляните на следующие объявления:

```
int hilt(int);
int halt(int) noexcept;
```

Выражение noexcept(hilt) вычисляется как false, поскольку объявление hilt() не гарантирует, что исключение не будет сгенерировано. Однако noexcept(halt) вычисляется как true.

E

Шаблонный класс `string`

Большая часть этого приложения посвящена техническим вопросам. Однако если вы желаете просто узнать о возможностях шаблонного класса `string`, можете ознакомиться только с описаниями различных методов `string`.

Класс `string` основан на таком определении шаблона:

```
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
class basic_string {...};
```

Здесь `charT` представляет тип, который хранится в строке. Параметр `traits` представляет класс, определяющий необходимые свойства, которыми должен обладать тип для представления строки. Например, он должен иметь метод `length()`, который возвращает длину строки, представленную в виде массива типа `charT`. Конец такого массива указан значением `charT(0)`, которое является обобщенной формой нулевого символа. (Выражение `charT(0)` — это приведение `0` к типу `charT`. Оно может быть равно просто `0`, как для типа `char`, или, в более общем случае, соответствовать объекту, созданному конструктором `charT`.) Класс также включает методы сравнения значений и т.п. Параметр `Allocator` представляет класс для управления распределением памяти под строку. Шаблон по умолчанию `allocator<charT>` использует операции `new` и `delete` стандартными способами.

Существуют четыре предварительно определенных специализации:

```
typedef basic_string<char> string;
typedef basic_string<char16_t> u16string;
typedef basic_string<char32_t> u32string;
typedef basic_string<wchar_t> wstring;
```

В свою очередь, эти специализации используют следующие специализации:

```
char_traits<char>
allocator<char>
char_traits<char16_t>
allocator<char_16>
```

```
char_traits<char_32>
allocator<char_32>
char_traits<wchar_t>
allocator<wchar_t>
```

Класс srting можно создать для типа, отличного от char или wchar_t, определяя класс traits и используя шаблон basic_string.

Тринадцать типов и константа

Шаблон basic_string определяет множество типов, которые могут применяться в определениях методов:

```
typedef traits                                  traits_type;
typedef typename traits::char_type              value_type;
typedef Allocator                               allocator_type;
typedef typename Allocator::size_type           size_type;
typedef typename Allocator::difference_type     difference_type;
typedef typename Allocator::reference           reference;
typedef typename Allocator::const_reference     const_reference;
typedef typename Allocator::pointer             pointer;
typedef typename Allocator::const_pointer       const_pointer;
```

Обратите внимание, что traits является шаблонным параметром, который соответствует одному из определенных типов; например, char_traits<char>; traits_type становится typedef для этого специфического типа. Следующая нотация означает, что char_type представляет собой имя типа, определенного в классе, который представлен traits:

```
typedef typename traits::char_type              value_type;
```

Ключевое слово typename служит для сообщения компилятору о том, что выражение traits::char_type представляет собой тип. Для специализации string, например, value_type имеет тип char.

size_type используется подобно size_of, за исключением того, что возвращает размер строки в терминах сохраняемого типа. Для специализации string это может быть тип char, в случае которого size_type эквивалентно size_of. Этот тип является беззнаковым.

difference_type служит для определения расстояния между двумя элементами строки, которое выражается в единицах, соответствующих размеру одного элемента. Обычно это версия со знаком типа на основе size_type.

Для специализации char тип pointer является типом char *, а reference — типом char &. Однако если создать специализацию для спроектированного типа, то эти типы (pointer и reference) могут относиться к классу, имеющему те же свойства, что и базовые указатели и ссылки.

Чтобы алгоритмы стандартной библиотеки шаблонов (Standard Template Library — STL) можно было использовать в строках, в шаблоне определены некоторые типы итераторов:

```
typedef (models random access iterator)         iterator;
typedef (models random access iterator)         const_iterator;
typedef std::reverse_iterator<iterator>         reverse_iterator;
typedef std::reverse_iterator<const_iterator>   const_reverse_iterator;
```

В шаблоне определена также и статическая константа:

```
static const size_type npos = -1;
```

Поскольку size_type является беззнаковым типом, то присваивание значения -1 в действительности будет соответствовать присваиванию npos наибольшего возможного значения без знака. Это значение соответствует значению, которое больше самого большого индекса массива.

Информация о данных, конструкторы и вспомогательные элементы

Конструкторы могут быть описаны в терминах оказываемых ими эффектов. Поскольку закрытые части класса могут зависеть от реализации, то эти эффекты должны описываться в терминах информации, доступной как часть открытого интерфейса. В табл. E.1 перечислены методы, возвращаемые значения которых могут использоваться для описания эффектов от конструкторов и других методов. Обратите внимание, что большинство терминологии взято из STL.

Таблица E.1. Некоторые методы работы с данными класса `string`

Метод	Возвращаемое значение
`begin()`	Итератор, указывающий на первый символ в строке
`cbegin()`	Итератор `const`, указывающий на первый символ в строке (C++11)
`end()`	Итератор, указывающий на элемент, следующий за последним
`cend()`	Итератор `const`, указывающий на элемент, следующий за последним (C++11)
`rbegin()`	Обратный итератор, указывающий на элемент, следующий за последним
`crbegin()`	Обратный итератор `const`, указывающий на элемент, следующий за последним (C++11)
`rend()`	Обратный итератор, указывающий на первый символ
`crend()`	Обратный итератор `const`, указывающий на первый символ (C++11)
`size()`	Количество элементов в строке, равное расстоянию от `begin()` до `end()`
`length()`	То же, что и `size()`
`capacity()`	Выделенное количество элементов в строке. Может быть больше действительного количества символов. Значение `capacity() - size()` представляет количество символов, которые могут быть присоединены к строке до того, как возникнет необходимость в выделении большего количества памяти
`max_size()`	Максимально допустимый размер строки
`data()`	Указатель типа `const charT*`, который указывает на первый элемент массива, чьи первые `size()` элементов равны соответствующим элементам в строке, управляемой `*this`. Указатель не должен считаться действительным после того, как был модифицирован сам объект `string`
`c_str()`	Указатель типа `const charT*`, который указывает на первый элемент массива, чьи первые `size()` элементов равны соответствующим элементам в строке, управляемой `*this`, и чей следующий элемент является символом `charT(0)` (маркер окончания строки) для типа `charT`. Указатель не должен считаться действительным после того, как был модифицирован сам объект `string`
`get_allocator()`	Копия объекта `allocator`, который используется для распределения памяти для объекта `string`

Необходимо понимать отличия между методами `begin()`, `rend()`, `data()` и `c_str()`. Все они связаны с первым символом в строке, но разными способами. Методы `begin()` и `rend()` возвращают итераторы, которые представляют собой обобщенную форму указателя, о чем говорилось в главе 16. В частности, метод `begin()` возвращает модель однонаправленного итератора, а `rend()` — копию обратного итератора. Оба метода относятся к действительной строке, управляемой объектом `string`. (Поскольку класс `string` использует динамическое распределение памяти, то действительное содержимое строки не должно находиться внутри объекта, поэтому для описания взаимосвязи между объектом и строкой применяется термин *управляемая*.) Методы, возвращающие итераторы, можно использовать с алгоритмами STL на основе итераторов. Например, функцию `reverse()` из STL можно применять обращения содержимого строки:

```
string word;
cin >> word;
reverse(word.begin(), word.end());
```

С другой стороны, методы `data()` и `c_str()` возвращают обычные указатели. Более того, возвращаемые указатели указывают на первый элемент *массива*, который содержит символы строки. Этот массив может быть, но не обязательно, копией исходной строки, управляемой объектом `string`. (Внутреннее представление объекта `string` может быть определено в виде массива, но это не обязательно.) Поскольку возвращаемые указатели могут указывать на исходные данные, то они имеют тип `const`, поэтому не могут применяться для изменения данных. Кроме этого, указатели могут стать недействительными после изменения строки, а, значит, они могут указывать на исходные данные. Различие между методами `data()` и `c_str()` заключается в том, что массив, на который указывает `c_str()`, завершается нулевым символом (или его эквивалентом), в то время как `data()` просто гарантирует наличие действительных символов строки. Таким образом, метод `c_str()` может использоваться, например, в качестве аргумента для функции, которая должна получить строку в стиле C:

```
string file("tofu.man");
ofstream outFile(file.c_str());
```

Подобным же образом методы `data()` и `size()` могут применяться вместе с функцией, которая должна получить указатель на элемент массива и значение, представляющее количество элементов для обработки:

```
string vampire("Do not stake me, oh my darling!");
int vlad = byte_check(vampire.data(), vampire.size());
```

В реализации C++ можно представить строку объекта `string` в виде динамически размещаемой строки в стиле C и реализовать прямой итератор как указатель `char *`. В этом случае реализация может обеспечить возврат методами `begin()`, `data()` и `c_str()` одного и того же указателя. Однако более предпочтительным и простым вариантом является возврат ссылок на три различных объекта данных.

В C++11 шаблонный класс `basic_string` имеет 11 конструкторов (в C++98 их было шесть) и один деструктор:

```
explicit basic_string(const Allocator& a = Allocator());
basic_string(const charT* s, const Allocator& a = Allocator());
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
basic_string(const basic_string& str);
basic_string(const basic_string& str, const Allocator&);
```

```
basic_string(const basic_string& str, size_type pos,
             size_type n = npos, const Allocator& a = Allocator());
basic_string(basic_string&& str) noexcept;
basic_string(const basic_string&& str, const Allocator&);
basic_string(size_type n, charT c, const Allocator& a = Allocator());
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
             const Allocator& a = Allocator());
basic_string(initializer_list<charT>, const Allocator& = Allocator());
~basic_string();
```

Некоторые из дополнительных конструкторов добавлены из-за различной обработки аргументов. Например, в C++98 определен следующий конструктор копирования:

```
basic_string(const basic_string& str, size_type pos = 0,
             size_type n = npos, const Allocator& a = Allocator());
```

В C++11 он заменен тремя конструкторами — второй, третий и четвертый элементы в предшествующем списке. Это позволяет более эффективно закодировать самое частое применение версии C++98. Действительно новыми дополнениями являются конструкторы переноса (со ссылками rvalue, как показано в главе 18) и конструктор с параметром `initializer_list`.

Обратите внимание, что большинство конструкторов принимают аргумент следующего вида:

```
const Allocator& a = Allocator()
```

Вспомните, что `Allocator` — это имя шаблонного параметра для класса `allocator`, который предназначен для управления памятью. `Allocator()` — это конструктор по умолчанию этого класса. Таким образом, по умолчанию конструкторы используют версию объекта `allocator`, предлагаемую по умолчанию, однако они дают возможность применять другую версию объекта `allocator`. В следующих разделах речь пойдет о каждом конструкторе отдельно.

Конструктор по умолчанию

Прототип для конструктора по умолчанию выглядит следующим образом:

```
explicit basic_string(const Allocator& a = Allocator());
```

Обычно вы будете принимать аргумент по умолчанию для класса `allocator` и использовать этот конструктор для создания пустых строк:

```
string bean;
wstring theory;
```

После вызова конструктора по умолчанию устанавливаются перечисленные ниже отношения.

- Метод `data()` возвращает ненулевой указатель, к которому может быть добавлено значение `0`.
- Метод `size()` возвращает `0`.
- Возвращаемое значение для `capacity()` не определено.

Предположим, что вы присваиваете значение, возвращаемое методом `data()`, указателю `str`. В этом случае первое условие означает, что `str + 0` является допустимым.

Конструкторы, использующие строки в стиле C

Конструкторы, использующие строки в стиле C, позволяют инициализировать объект `string` строкой в стиле C; в общем случае они позволяют инициализировать специализацию `charT` с помощью массива значений `charT`:

```
basic_string(const charT* s, const Allocator& a = Allocator());
```

Чтобы определить, сколько необходимо скопировать символов, конструктор применяет метод `traits::length()` к массиву, на который указывает `s`. (Указатель `s` не должен быть нулевым.) Например, следующий оператор инициализирует объект `toast`, используя указанную строку символов:

```
string toast("Here's looking at you, kid.");
```

Метод `traits::length()` для типа `char` использует нулевой символ, чтобы определить количество символов, которые необходимо скопировать.

После вызова конструктора устанавливаются следующие отношения.

- Метод `data()` возвращает указатель на первый элемент копии массива `s`.
- Метод `size()` возвращает значение, равное `traits::length()`.
- Метод `capacity()` возвращает значение, которое как минимум такое же большое, как и `size()`.

Конструкторы, использующие часть строки в стиле C

Конструкторы, использующие часть строки в стиле C, позволяют инициализировать объект `string` частью строки в стиле C; в общем случае они позволяют инициализировать специализацию `charT` с помощью части массива значений `charT`:

```
basic_string(const charT* s, size_type n, const Allocator& a = Allocator());
```

Этот конструктор копирует в создаваемый объект всего `n` символов из массива, на который указывает `s`. Обратите внимание, что копирование будет продолжаться до тех пор, пока указатель `s` будет иметь меньшее количество символов, чем `n`. Если `n` превышает длину `s`, то конструктор интерпретирует содержимое за строкой так, как будто там содержатся данные типа `charT`.

Данный конструктор требует, чтобы указатель `s` не был нулевым и `n < npos`. (Вспомните, что `npos` является статической константой класса, равной максимально возможному количеству элементов в строке.) Если `n` будет равно `npos`, конструктор сгенерирует исключение `out_of_range`. (Поскольку `n` имеет тип `size_type`, а `npos` является максимальным значением `size_type`, `n` не может быть больше `npos`.) Иначе после вызова конструктора будут установлены следующие отношения.

- Метод `data()` возвращает указатель на первый элемент копии массива `s`.
- Метод `size()` возвращает `n`.
- Метод `capacity()` возвращает значение, которое как минимум такое же большое, как и `size()`.

Конструкторы, использующие ссылку lvalue

Конструктор копирования выглядит следующим образом:

```
basic_string(const basic_string& str);
```

Он инициализирует новый объект string с использованием аргумента string:

```
string mel("I'm ok!");
string ida(mel);
```

Здесь ida получит копию строки, управляемой mel.

Следующий конструктор дополнительно требует указания объекта для управления распределением памяти:

```
basic_string(const basic_string& str, const Allocator&);
```

После вызова любого из упомянутых конструкторов будут установлены перечисленные ниже отношения.

- Метод data() возвращает указатель на выделенную копию массива, первый элемент которого указан с помощью str.data().
- Метод size() возвращает значение str.size().
- Метод capacity() возвращает значение, которое как минимум такое же большое, как и size().

Следующий конструктор позволяет установить несколько элементов:

```
basic_string(const basic_string& str, size_type pos, size_type n = npos,
             const Allocator& a = Allocator());
```

Второй аргумент pos определяет позицию в исходной строке, начиная с которой будет производиться копирование:

```
string att("Telephone home.");
string et(att, 4);
```

Номера позиций начинаются с 0, поэтому позиция 4 соответствует символу p. Таким образом, et инициализируется строкой "phone home.".

Необязательный третий аргумент n задает максимальное количество символов, которые будут скопированы. Таким образом, приведенный ниже код инициализирует pt строкой "phone":

```
string att("Telephone home.");
string pt(att, 4, 5);
```

Однако этот конструктор не выходит за пределы исходной строки; например, следующий вызов останавливается после того, как будет скопирована точка:

```
string pt(att, 4, 200)
```

Таким образом, в действительности конструктор копирует количество символов, равное меньшему из значений n и str.size() - pos.

Данный конструктор требует выполнения условия pos <= str.size(), т.е. чтобы исходная позиция, с которой начнется копирование, находилась в пределах исходной строки; если это условие нарушается, конструктор генерирует исключение out_of_range. В противном случае, если copy_len будет представлять меньше из значений n и str.size() - pos, то после вызова конструктора будут установлены следующие отношения.

- Метод data() возвращает указатель на копию copy_len элементов, скопированных из строки str, начиная с позиции pos внутри str.
- Метод size() возвращает copy_len.
- Метод capacity() возвращает значение, которое как минимум такое же большое, как и size().

Конструкторы, использующие ссылку rvalue (C++11)

C++11 добавляет к классу `string` семантику переноса. Как показано в главе 18, это приводит к добавлению конструктора переноса, который использует вместо ссылки lvalue ссылку rvalue:

```
basic_string(basic_string&& str) noexcept;
```

Этот конструктор вызывается, когда действительный аргумент является временным объектом:

```
string one("din");              // конструктор, использующий строку в стиле C
string two(one);                // конструктор копирования; one – это lvalue
string three(one+two);          // конструктор переноса; сумма – это rvalue
```

Как объяснялось в главе 18, смысл заключается в том, что строка `three` получает право владения объектом, который сконструирован с помощью `operator+()`, вместо того, чтобы копировать этот объект и затем позволить его уничтожить.

Второй конструктор, использующий ссылку rvalue, дополнительно позволяет указать объект для управления распределением памяти:

```
basic_string(const basic_string&& str, const Allocator&);
```

После вызова любого из этих двух конструкторов устанавливаются следующие отношения.

- Метод `data()` возвращает указатель на выделенную копию массива, первый элемент которого указан с помощью `str.data()`.
- Метод `size()` возвращает значение `str.size()`.
- Метод `capacity()` возвращает значение, которое как минимум такое же большое, как и `size()`.

Конструктор, использующий `n` копий символа

Конструктор, использующий `n` копий символа, создает объект `string`, состоящий из `n` последовательных символов, каждый из которых имеет значение `c`:

```
basic_string(size_type n, charT c, const Allocator& a = Allocator());
```

Для конструктора необходимо, чтобы удовлетворялось условие `n < npos`. Если `n` будет равно `npos`, конструктор сгенерирует исключение `out_of_range`. В противном случае после вызова конструктора будут установлены следующие отношения.

- Метод `data()` возвращает указатель на первый элемент строки, состоящей из `n` элементов, каждый из которых имеет символ `c`.
- Метод `size()` возвращает `n`.
- Метод `capacity()` возвращает значение, которое как минимум такое же большое, как и `size()`.

Конструктор, использующий диапазон

Этот конструктор использует диапазон, определяемый итератором в стиле STL:

```
template<class InputIterator>
basic_string(InputIterator begin, InputIterator end,
             const Allocator& a = Allocator());
```

Итератор begin указывает на элемент в исходной строке, с которого начнется копирование, а end — на последнюю позицию, которая будет скопирована. Эту форму конструктора можно применять для массивов, строк или контейнеров STL:

```
char cole[40] = "Old King Cole was a merry old soul.";
string title(cole + 4, cole + 8);
vector<char> input;
char ch;
while (cin.get(ch) && ch != '\n')
   input.push_back(ch);
string str_input(input.begin(), input.end());
```

При первом использовании InputIterator вычисляется как тип const char *.

При втором использовании InputIterator вычисляется как тип vector<char>::iterator. После вызова этого конструктора устанавливаются перечисленные ниже отношения.

- Метод data() возвращает указатель на первый элемент строки, сформированной посредством копирования элементов из диапазона [begin, end).
- Метод size() возвращает расстояние между begin и end. (Расстояние измеряется в единицах, которые равны размеру типа данных, полученному при разыменовании итератора.)
- Метод capacity() возвращает значение, которое как минимум такое же большое, как и size().

Конструктор, использующий список инициализаторов (C++11)

Этот конструктор получает параметр initializer_list<charT>:

```
basic_string(initializer_list<charT> il, const Allocator& a = Allocator());
```

Его можно использовать со списком символов в фигурных скобках:

```
string slow({'s', 'n', 'a', 'i', 'l'});
```

Хотя это не самый удобный способ инициализации строки, он сохраняет интерфейс string подобным тому, который используется контейнерными классами STL.

Класс initializer_list имеет члены begin() и end(). Результат применения этого конструктора будет таким же, как и конструктора, использующего диапазон:

```
basic_string(il.begin(), il.end(), a);
```

Различные действия с памятью

Работа некоторых методов связана с памятью, а именно — с очисткой содержимого памяти, изменением размеров строки и настройкой вместимости строки. В табл. E.2 перечислены методы, работа которых связана с памятью.

Доступ к строке

Существуют четыре способа доступа к индивидуальным символам, два из которых используют операцию [], а два других — метод at():

```
reference operator[](size_type pos);
const_reference operator[](size_type pos) const;
reference at(size_type n);
const_reference at(size_type n) const;
```

Таблица E.2. Некоторые методы, работа которых связана с памятью

Метод	Результат выполнения
`void resize (size_type n)`	Генерирует исключение `out_of_range`, если `n > pos`. В противном случае изменяет размер строки до `n`, отбрасывая конец строки, если `n < size()`, и заполняя строку символами `charT(0)`, если `n > size()`
`void resize(size_type n, charT c)`	Генерирует исключение `out_of_range`, если `n > npos`. В противном случае изменяет размер строки до `n`, отбрасывая конец строки, если `n < size()`, и заполняя строку символами `c`, если `n > size()`
`void reserve(size_type res_arg = 0)`	Устанавливает вместимость строки больше или равной `res_arg`. Поскольку при этом происходит повторное размещение строки, то предыдущие ссылки, итераторы и указатели на строку аннулируются
`void shrink_to_fit()`	Несвязанный запрос для уменьшения вместимости строки до `size()` (C++11)
`void clear() noexcept`	Удаляет все символы из строки
`bool empty() const noexcept`	Возвращает `true`, если `size() == 0`

Первый метод `operator[]()` позволяет обращаться к индивидуальному элементу строки, используя нотацию массива; этот вариант можно применять для получения или изменения значения. Второй метод `operator[]()` можно использовать вместе с объектами `const`, и он предназначен только для получения значения:

```
string word("tack");
cout << word[0];          // отображает t
word[3] = 't';            // перезаписывает k символом t
const ward("garlic");
cout << ward[2];          // отображает r
```

Методы `at()` предлагают похожую схему доступа, за исключением того, что индекс предоставляется как аргумент функции:

```
string word("tack");
cout << word.at(0);       // отображает t
```

Различие между ними, помимо различия в синтаксисе, заключается в том, что методы `at()` обеспечивают проверку границ и генерируют исключение `out_of_range`, если `pos >= size()`. Обратите внимание, что `pos` имеет тип `size_type`, который является беззнаковым; таким образом, для `pos` отрицательные значения не допускаются. Методы `operator[]()` не выполняют проверку границ, поэтому их поведение будет неопределенным, если `pos >= size()`, кроме тех случаев, когда версия `const` возвращает эквивалент нулевого символа при условии, что `pos == size()`.

Итак, у вас имеется возможность выбора между безопасной работой (использование `at()` и проверка исключений) и скоростью выполнения (применение нотации массива).

Существует также функция, которая возвращает новую строку, являющуюся подстрокой исходной строки:

```
basic_string substr(size_type pos = 0, size_type n = npos) const;
```

Она возвращает строку, которая является копией исходной строки, начиная с позиции pos и включая n символов, или до конца строки — смотря, что наступит раньше. Например, в следующем фрагменте кода pet присваивается подстрока "donkey":

```
string message("Maybe the donkey will learn to sing.");
string pet(message.substr(10, 6));
```

В C++11 добавлены следующие четыре метода доступа:

```
const charT& front() const;
charT& front();
const charT& back() const;
charT& back();
```

Методы front() получают доступ к первому элементу строки, действуя подобно operator[](0). Методы back() получают доступ к первому элементу строки, действуя подобно operator[](size() - 1).

Базовое присваивание

C++ имеет пять перегруженных метода присваивания (по сравнению с тремя в C++98):

```
basic_string& operator=(const basic_string& str);
basic_string& operator=(const charT* s);
basic_string& operator=(charT c);
basic_string& operator=(basic_string&& str) noexcept;      // C++11
basic_string& operator=(initializer_list<charT>);          // C++11
```

Первый метод присваивает один объект string другому, второй присваивает строку в стиле C объекту string, третий присваивает одиночный символ объекту string, четвертый использует семантику переноса для присваивания rvalue-объекта string объекту string, а пятый позволяет выполнить присваивание с использованием списка инициализаторов. Таким образом, возможны следующие действия:

```
string name("George Wash");
string pres, veep, source, join, awkward;
pres = name;
veep = "Road Runner";
source = 'X';
join = name + source;  // теперь доступна семантика переноса
awkward = {'C','l','o','u','s','e','a','u'};
```

Поиск в строках

Класс string предоставляет шесть функций поиска, каждая из которых имеет четыре прототипа. Эти функции кратко описаны в последующих разделах.

Семейство find()

Ниже приведены прототипы find(), как они определены в C++11:

```
size_type find (const basic_string& str, size_type pos = 0) const noexcept;
size_type find (const charT* s, size_type pos = 0) const;
size_type find (const charT* s, size_type pos, size_type n) const;
size_type find (charT c, size_type pos = 0) const noexcept;
```

Первый член возвращает начальную позицию первого вхождения подстроки str в вызываемом объекте, при этом поиск начинается с позиции pos. Если подстрока не найдена, метод возвращает npos.

Далее показан пример поиска позиции подстроки "hat" в строке longer:

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find(shorter);          // устанавливает loc1 в 1
size_type loc2 = longer.find(shorter, loc1 + 1); // устанавливает loc2 в 16
```

Поскольку второй поиск начинается с позиции 2 (буква a в слове That), то первое вхождение подстроки hat будет найдено ближе к концу строки. Для проверки на неудачу используется значение string::npos:

```
if (loc1 == string::npos)
   cout << "Not found\n";
```

Второй метод делает то же самое, за исключением того, что в качестве подстроки он использует массив символов, а не объект string:

```
size_type loc3 = longer.find("is");             // устанавливает loc3 в 5
```

Третий метод выполняет то же самое, что и второй, за исключением того, что он использует только первые n символов строки s. Результат будет таким же, как и в случае применения конструктора basic_string(const charT* s, size_type n) и передачи результирующего объекта в качестве аргумента string первой форме find(). Например, следующий код ищет подстроку "fun":

```
size_type loc4 = longer.find("funds", 3);       // устанавливает loc4 в 10
```

Четвертый метод делает то же самое, что и первый, за исключением того, что в качестве подстроки в нем используется одиночный символ, а не объект string:

```
size_type loc5 = longer.find('a');              // устанавливает loc5 в 2
```

Семейство rfind()

Методы rfind() имеют следующие прототипы:

```
size_type rfind(const basic_string& str,
                size_type pos = npos) const noexcept;
size_type rfind(const charT* s, size_type pos = npos) const;
size_type rfind(const charT* s, size_type pos, size_type n) const;
size_type rfind(charT c, size_type pos = npos) const noexcept;
```

Эти методы работают аналогично методам find(), за исключением того, что они ищут последнее вхождение строки или символа, которое начинается или находится перед позицией pos. Если подстрока не найдена, метод возвращает npos.

Ниже приведен пример поиска подстроки "hat" в строке longer, начиная с ее конца:

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.rfind(shorter);         // устанавливает loc1 в 16
size_type loc2 = longer.rfind(shorter, loc1 - 1); // устанавливает loc2 в 1
```

Семейство find_first_of()

Методы find_first_of() имеют следующие прототипы:

```
size_type find_first_of(const basic_string& str,
                        size_type pos = 0) const noexcept;
size_type find_first_of(const charT* s, size_type pos, size_type n) const;
size_type find_first_of(const charT* s, size_type pos = 0) const;
size_type find_first_of(charT c, size_type pos = 0) const noexcept;
```

Эти методы работают подобно соответствующим методам find(), за исключением того, что вместо поиска совпадения со всей подстрокой они ищут первое совпадение с любым одиночным символом из подстроки:

```
string longer("That is a funny hat.");
string shorter("fluke");
size_type loc1 = longer.find_first_of(shorter);  // устанавливает loc1 в 10
size_type loc2 = longer.find_first_of("fat");    // устанавливает loc2 в 2
```

Первым вхождением любого из пяти символов "fluke" в строке longer является f в funny. Первым вхождением любого из трех символов "fat" в строке longer является a в That.

Семейство find_last_of()

Методы find_last_of() имеют следующие прототипы:

```
size_type find_last_of(const basic_string& str,
                       size_type pos = npos) const noexcept;
size_type find_last_of(const charT* s, size_type pos, size_type n) const;
size_type find_last_of(const charT* s, size_type pos = npos) const;
size_type find_last_of(charT c, size_type pos = npos) const noexcept;
```

Эти методы работают подобно соответствующим методам rfind(), за исключением того, что вместо поиска совпадения со всей подстрокой они ищут последнее совпадение с любым одиночным символом из подстроки.

Ниже приведен пример поиска вхождений любого из символов "hat" и "any" в строке longer:

```
string longer("That is a funny hat.");
string shorter("hat");
size_type loc1 = longer.find_last_of(shorter);   // устанавливает loc1 в 18
size_type loc2 = longer.find_last_of("any");     // устанавливает loc2 в 17
```

Последним вхождением любого из трех символов "fat" в строке longer является t в hat. Последним вхождением любого из трех символов "any" в строке longer является a в hat.

Семейство find_first_not_of()

Методы find_first_not_of() имеют следующие прототипы:

```
size_type find_first_not_of(const basic_string& str,
                            size_type pos = 0) const noexcept;
size_type find_first_not_of(const charT* s, size_type pos,
                            size_type n) const;
size_type find_first_not_of(const charT* s, size_type pos = 0) const;
size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
```

Эти методы работают подобно соответствующим методам `find_first_of()`, за исключением того, что они ищут первое вхождение любого символа, который отсутствует в подстроке.

Ниже приведен пример поиска в строке `longer` позиций первых вхождений символов, не совпадающих с символами в `"This"` и `"Thatch"`:

```
string longer("That is a funny hat.");
string shorter("This");
size_type loc1 = longer.find_first_not_of(shorter); // устанавливает loc1 в 2
size_type loc2 = longer.find_first_not_of("Thatch"); // устанавливает loc2 в 4
```

Буква `a` в слове `That` является первым символом в строке `longer`, который отсутствует в `This`. Первый пробел в строке `longer` является первым символом, который не присутствует в `Thatch`.

Семейство `find_last_not_of()`

Методы `find_last_not_of()` имеют следующие прототипы:

```
size_type find_last_not_of(const basic_string& str,
                           size_type pos = npos) const noexcept;
size_type find_last_not_of(const charT* s, size_type pos,
                           size_type n) const;
size_type find_last_not_of(const charT* s,
                           size_type pos = npos) const;
size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;
```

Эти методы работают подобно соответствующим методам `find_last_of()`, за исключением того, что они ищут последнее вхождение любого символа, который отсутствует в подстроке.

Ниже приведен пример поиска в строке `longer` позиций последних двух вхождений символов, не совпадающих с символами в `"That"`:

```
string longer("That is a funny hat.");
string shorter("That.");
size_type loc1 = longer.find_last_not_of(shorter);  //устанавливает loc1 в 15
size_type loc2 = longer.find_last_not_of(shorter, 10); //устанавливает loc2 в 10
```

Последний пробел в строке `longer` является последним символом, которого нет в строке `shorter`. Буква `f` в строке `longer` является последним символом, которого нет в строке `shorter` вплоть до позиции 10.

Методы и функции сравнения

Класс `string` предлагает методы и функции для сравнения двух строк. Для начала рассмотрим прототипы методов:

```
int compare(const basic_string& str) const noexcept;
int compare(size_type pos1, size_type n1,
          const basic_string& str) const;
int compare(size_type pos1, size_type n1,
          const basic_string& str,
          size_type pos2, size_type n2) const;
int compare(const charT* s) const;
int compare(size_type pos1, size_type n1, const charT* s) const;
int compare(size_type pos1, size_type n1,
          const charT* s, size_type n2 ) const;
```

Эти методы используют метод traits::compare(), который определяется для конкретного символьного типа, применяемого в строке. Первый метод возвращает значение меньше нуля, если первая строка предшествует второй строке в соответствии с порядком, заданным с помощью traits::compare(). Он возвращает 0, если две строки являются одинаковыми, и значение больше нуля, если за первой строкой будет следовать вторая. Если две строки идентичны вплоть до конца самой короткой из них, то короткая строка будет предшествовать длинной.

В следующем примере сравниваются строки s1 с s3 и s1 с s2:

```
string s1("bellflower");
string s2("bell");
string s3("cat");
int a13 = s1.compare(s3);              // a13 < 0
int a12 = s1.compare(s2);              // a12 > 0
```

Второй метод подобен первому, за исключением того, что сравнение производится с использованием n1 символов, начиная с позиции pos1 в первой строке.

В следующем примере сравниваются первые четыре символа в строке s1 с s2:

```
string s1("bellflower");
string s2("bell");
int a2 = s1.compare(0, 4, s2);         // a2 = 0
```

Третий метод подобен первому, за исключением того, что сравнение производится с использованием n1 символов, начиная с позиции pos1 в первой строке, и n2 символов, начиная с позиции pos2 во второй строке. Например, в следующем фрагменте кода сравнивается подстрока out в stout с подстрокой out в about:

```
string st1("stout boar");
string st2("mad about ewe");
int a3 = st1.compare(2, 3, st2, 6, 3);  // a3 = 0
```

Четвертый метод подобен первому, за исключением того, что в нем для второй строки применяется массив символов, а не объект string.

Пятый и шестой методы подобны третьему, за исключением того, что в них для второй строки используется массив символов, а не объект string.

Функции сравнения, не являющиеся членами, представляют собой перегруженные операции отношения:

```
operator==()
operator<()
operator<=()
operator>()
operator>=()
operator!=()
```

Каждая операция перегружается, чтобы можно было сравнивать объект string с объектом string, объект string со строкой в стиле C и строку в стиле C с объектом string. Эти операции определены на основе метода compare(), поэтому они гораздо удобнее с точки зрения обозначений.

Модификация строк

Класс string предлагает несколько методов модификации строк. Большинство из них имеет множество перегруженных версий, поэтому они могут использоваться для объектов string, массивов строк, индивидуальных символов и диапазонов итераторов.

Методы присоединения и добавления

Для присоединения одной строки к другой можно использовать перегруженную операцию += или метод append(). Оба они генерируют исключение length_error, если результат будет больше максимального размера строки. С помощью операции += можно присоединить объект string, массив строк или индивидуальный символ к другой строке:

```
basic_string& operator+=(const basic_string& str);
basic_string& operator+=(const charT* s);
basic_string& operator+=(charT c);
```

С помощью методов append() можно присоединять объект string, массив строк либо индивидуальный символ к другой строке. Кроме этого, они позволяют присоединить часть объекта string за счет определения начальной позиции и количества присоединяемых символов либо же определения их диапазона. Можно присоединить часть строки, указав необходимое количество символов. Версия с присоединением символа позволяет определить, сколько будет скопировано экземпляров этого символа. Ниже приведены прототипы различных методов append():

```
basic_string& append(const basic_string& str);
basic_string& append(const basic_string& str, size_type pos,
                     size_type n);
template<class InputIterator>
   basic_string& append(InputIterator first, InputIterator last);
basic_string& append(const charT* s);
basic_string& append(const charT* s, size_type n);
basic_string& append(size_type n, charT c); // присоединяет n копий символа c
void push_back(charT c);                    // присоединяет 1 копию символа c
```

Вот пара примеров:

```
string test("The");
test.append("ory");           // строка test хранит "Theory"
test.append(3, '!');          // строка test хранит "Theory!!!"
```

Функция operator+() перегружается, чтобы сделать возможной конкатенацию строк. Перегруженные функции не изменяют строку; наоборот, они создают новую строку, состоящую из одной строки, к которой присоединена другая. Функции добавления не являются функциями-членами и позволяют добавлять объект string к объекту string, массив строк к объекту string и объект string к символу. Ниже показаны некоторые примеры:

```
string st1("red");
string st2("rain");
string st3 = st1 + "uce";      // строка st3 хранит "reduce"
string st4 = 't' + st2;        // строка st4 хранит "train"
string st5 = st1 + st2;        // строка st5 хранит "redrain"
```

Дополнительные методы присваивания

Кроме базовой операции присваивания класс string предлагает методы assign(), посредством которых можно присваивать объекту string всю строку целиком, часть строки или последовательность одинаковых символов. Ниже представлены прототипы различных методов assign():

```
basic_string& assign(const basic_string& str);
basic string& assign(basic_string&& str) noexcept;  // C++11
```

```
basic_string& assign(const basic_string& str, size_type pos,
                     size_type n);
basic_string& assign(const charT* s, size_type n);
basic_string& assign(const charT* s);
basic_string& assign(size_type n, charT c); // присваивает n копий символа c
template<class InputIterator>
   basic_string& assign(InputIterator first, InputIterator last);
basic_string& assign(initializer_list<charT>);   // C++11
```

Вот некоторые примеры:

```
string test;
string stuff("set tubs clones ducks");
test.assign(stuff, 1, 5);                    // строка test хранит "et tu"
test.assign(6, '#");                         // строка test хранит "######"
```

Метод `assign()` со ссылкой rvalue (появившийся в C++11) делает возможной семантику переноса, а второй новый метод `assign()` позволяет присвоить `initializer_list` объекту `string`.

Методы вставки

С помощью методов вставки `insert()` можно вставить в объект `string` другой объект `string`, массив строк, символ или несколько символов. Эти методы подобны методам `append()`, за исключением того, что они принимают дополнительный аргумент, указывающий позицию, куда будет произведена вставка нового материала. Этот аргумент может быть представлен позицией или итератором. Информация вставляется перед точкой вставки. Некоторые методы возвращают ссылку на результирующую строку. Если `pos1` будет находиться за пределами искомой строки или если `pos2` будет находиться за пределами вставляемой строки, метод сгенерирует исключение `out_of_range`. Если размер результирующей строки окажется больше максимального, метод сгенерирует исключение `length_error`. Ниже показаны прототипы различных методов `insert()`:

```
basic_string& insert(size_type pos1, const basic_string& str);
basic_string& insert(size_type pos1, const basic_string& str,
                     size_type pos2, size_type n);
basic_string& insert(size_type pos, const charT* s, size_type n);
basic_string& insert(size_type pos, const charT* s);
basic_string& insert(size_type pos, size_type n, charT c);
iterator insert(const_iterator p, charT c);
iterator insert(const_iterator p, size_type n, charT c);
template<class InputIterator>
   void insert(iterator p, InputIterator first, InputIterator last);
iterator insert(const_iterator p, initializer_list<charT>);  // C++11
```

Например, следующий код вставляет строку `"former "` перед буквой `b` в строке `"The banker."`:

```
string st3("The banker.");
st3.insert(4, "former ");
```

Следующий код вставляет строку `" waltzed"` (не включая символ `!`, который будет девятым символом) непосредственно перед точкой в конце строки `"The former banker."`:

```
st3.insert(st3.size() - 1, " waltzed!", 8);
```

Методы очистки

Методы очистки `erase()` удаляют символы из строки. Ниже представлены их прототипы:

```
basic_string& erase(size_type pos = 0, size_type n = npos);
iterator erase(const_iterator position);
iterator erase(const_iterator first, iterator last);
void pop_back();
```

Первая форма перемещает символ из позиции `pos` на `n` символов далее или в конец строки — смотря, что наступит раньше. Вторая форма удаляет одиночный символ, на который ссылается позиция итератора, и возвращает итератор на следующий элемент, а если элементов больше нет, то возвращает `end()`. Третья форма удаляет символы в диапазоне [`first`, `last`), т.е. включая `first` и не включая `last`. Метод возвращает итератор на элемент, который следует за последним удаленным символом. Наконец, метод `pop_back()` удаляет последний символ строки.

Методы замены

Различные методы `replace()` идентифицируют часть строки, которая должна быть заменена, а также определяют замену. Заменяемую часть можно идентифицировать по начальной позиции и счетчику символов или диапазоном итератора. В качестве замены может выступать объект `string`, массив строк либо отдельный символ, дублированный несколько раз. Объекты `string` и массивы впоследствии можно модифицировать, указывая определенную часть, используя позицию и счетчик, просто счетчик или диапазон итератора. Ниже представлены прототипы различных методов `replace()`:

```
basic_string& replace(size_type pos1, size_type n1, const basic_string& str);
basic_string& replace(size_type pos1, size_type n1, const basic_string& str,
                      size_type pos2, size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s,
                      size_type n2);
basic_string& replace(size_type pos, size_type n1, const charT* s);
basic_string& replace(size_type pos, size_type n1, size_type n2, charT c);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const basic_string& str);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const charT* s, size_type n);
basic_string& replace(const_iterator i1, const_iterator i2,
                      const charT* s);
basic_string& replace(const_iterator i1, const_iterator i2,
                      size_type n, charT c);
template<class InputIterator>
basic_string& replace(const_iterator i1, const_iterator i2,
                      InputIterator j1, InputIterator j2);
basic_string& replace(const_iteraor i1, const_iteator i2,
                      initializer)list<charT> il);
```

А вот пример:

```
string test("Take a right turn at Main Street.");
test.replace(7, 5, "left");  // заменяет слово right словом left
```

Для поиска позиций, используемых в replace, можно применить find():

```
string s1 = "old";
string s2 = "mature";
string s3 = "The old man and the sea";
string::size_type pos = s3.find(s1);
if (pos != string::npos)
   s3.replace(pos, s1.size(), s2);
```

В этом примере слово old будет заменено словом mature.

Другие методы модификации: copy() и swap()

Метод copy() копирует объект string или его часть в целевой массив символов:

```
size_type copy(charT* s, size_type n, size_type pos = 0) const;
```

В этом случае s указывает на искомый массив, n задает количество копируемых символов, а pos определяет позицию в объекте string, с которой начнется копирование. Будут скопированы n символов или символы вплоть до последнего в объекте string — смотря, что наступит раньше. Функция возвращает количество скопированных символов. Метод не присоединяет нулевой символ, и программист должен самостоятельно определить, может ли массив уместить скопированные символы.

> **Внимание!**
>
> Метод copy() не присоединяет нулевой символ и не проверяет, может ли искомый массив уместить скопированные символы.

Метод swap() производит обмен содержимого двух объектов string с помощью алгоритма с константным временем:

```
void swap(basic_string& str);
```

Ввод и вывод

Для отображения объектов string класс string перегружает операцию <<. Она возвращает ссылку на объект istream, поэтому можно осуществлять конкатенацию вывода:

```
string claim("The string class has many features.");
cout << claim << endl;
```

Класс string перегружает операцию >>, поэтому можно считывать ввод в строку:

```
string who;
cin >> who;
```

Ввод прекращается по достижении конца файла, когда прочитано максимальное количество символов, которые может уместить строка, или при встрече с пробельным символом. (Определение пробельного символа зависит от набора символов и типа, который представляет charT.)

Доступны две функции getline(). Первая имеет следующий прототип:

```
template<class charT, class traits, class Allocator>
   basic_istream<charT,traits>& getline(basic_istream<charT,traits>& is,
                basic_string<charT,traits,Allocator>& str, charT delim);
```

Она считывает символы из потока ввода is в строку str, пока не будет достигнут символ-ограничитель delim, максимальный размер строки или конец файла. Символ delim читается (т.е. удаляется из потока ввода), но не сохраняется. Во втором варианте отсутствует третий аргумент, а вместо delim используется символ новой строки (либо его обобщенная форма):

```
string str1, str2;
getline(cin, str1);           // чтение до конца строки
getline(cin, str2, '.');      // чтение до символа точки
```

Ж

Методы и функции стандартной библиотеки шаблонов

Стандартная библиотека шаблонов (Standard Template Library – STL) содержит эффективные реализации распространенных алгоритмов. Она представляет их в виде общих функций, которые могут использоваться с любым контейнером, удовлетворяющим требованиям к определенному алгоритму, а также в виде методов, которые могут применяться в реализациях определенных классов контейнеров. В этом приложении предполагается, что вы уже имеете некоторое представление о библиотеке STL. Для начала вы должны ознакомиться с материалом главы 16. Например, вам должны быть знакомы понятия итераторов и конструкторов.

Библиотека STL и C++11

Подобно тому, как изменения, внесенные в язык стандартом C++11, слишком обширны, чтобы их можно было рассмотреть полностью в этой книге, изменения в STL невозможно уместить полностью в одно приложение. Тем не менее, мы кратко опишем основные дополнения.

Версия C++11 привнесла в STL множество новых элементов. Во-первых, добавлены полностью новые контейнеры. Во-вторых, старые контейнеры получили несколько новых функциональных возможностей. В-третьих, семейства алгоритмов пополнены новыми шаблонными функциями. Хотя все эти изменения отражены в данном приложении, полезно сначала ознакомиться с обзором первых двух их категорий.

Новые контейнеры

C++11 добавляет обычные контейнеры `array`, `forward_list`, `unordered_set`, а также неупорядоченные ассоциативные контейнеры `unordered_multiset`, `unordered_map` и `unordered_multimap`.

Контейнер array после объявления имеет фиксированный размер и работает со статической памятью или стеком, а не с динамически выделяемой памятью. Он предназначен служить заменой встроенного типа массива; он более ограничен, чем vector, но гораздо эффективнее.

Контейнер list – это двухсвязный список, в котором каждый элемент кроме двух крайних связан с предшествующим и последующим элементами.

Контейнер forward_list – это односвязный список, в котором каждый элемент кроме последнего связан с последующим элементом. Он является более компактной, но ограниченной альтернативой контейнеру list.

Подобно set и другим ассоциативным контейнерам, неупорядоченные ассоциативные контейнеры позволяют быстро извлекать данные с использованием ключей. Отличие состоит в том, что обычные ассоциативные контейнеры используют в качестве лежащей в основе структуры данных деревья, в то время как неупорядоченные ассоциативные контейнеры – хеш-таблицы.

Изменения в контейнерах C++98

C++11 привносит три основных изменения в методы контейнерных классов.

Во-первых, добавление ссылок rvalue делает возможной реализацию семантики переноса (глава 18) для контейнеров. Соответственно, библиотека STL теперь предоставляет для контейнеров конструкторы переноса и операции присваивания с переносом. Эти метода принимают аргумент – ссылку rvalue.

Во-вторых, добавление шаблонного класса initializer_list (глава 18) привело к появлению конструкторов и операций присваивания, которые принимают аргумент initializer_list. Это делает возможным код следующего вида:

```
vector<int> vi{100, 99, 97, 98};
vi = {96, 99, 94, 95, 102};
```

В-третьих, добавление шаблонов с переменным числом аргументов и пакетов параметров функций (глава 18) делает возможными заменяющие методы. Что это означает? Подобно семантике переноса, замена (emplacement) означает увеличение эффективности. Предположим, что имеется следующие фрагменты кода:

```
class Items
{
   double x;
   double y;
   int m;
public:
   Items();                                    // #1
   Items(double xx, double yy, int mm);        // #2
...
};
...
vector<Items> vt(10);
...
vt.push_back(Items(8.2, 2.8, 3));
```

Вызов insert() приводит к тому, что функция выделения памяти создает стандартный объект Items в конце vt. Затем конструктор Items() создает временный объект Items; этот объект копируется в позицию в начале вектора vt, после чего временный объект удаляется. В C++11 вместо этого можно поступить следующим образом:

```
vi.emplace_back(8.2, 2.8, 3);
```

Метод emplace_back() — это шаблон с переменным числом аргументов, в качестве аргумента которого выступает пакет параметров функции:

```
template <class... Args> void emplace_back(Args&&... args);
```

Три аргумента 8.2, 2.8 и 3 упакованы в параметр args. Эти параметры передаются вместе с функцией выделения памяти, которая затем распаковывает их и применяет конструктор Items с тремя аргументами (#2) вместо конструктора по умолчанию (#1). То есть он может использовать конструктор Items(args...), который в данном примере расширяется до Items(8.2, 2.8, 3). Таким образом, требуемый объект конструируется на месте в векторе, а не где-то во временной ячейке с последующим копированием его в вектор. Описанный прием применяется в STL со многими заменяющими методами.

Члены, общие для всех или большинства контейнеров

Все контейнеры определяют типы, перечисленные в табл. Ж.1. В этой таблице X — тип контейнера, например, vector<int>, а T — тип, хранящийся в контейнере, такой как int. Примеры, следующие за таблицей, помогают понять назначение контейнеров.

Таблица Ж.1. Типы, определенные для всех контейнеров

Тип	Значение
X::value_type	T, тип элемента
X::reference	T &
X::const_reference	const T &
X::iterator	Тип итератора, указывающего на T; его поведение подобно типу T *
X::const_iterator	Тип итератора, указывающего на const T; его поведение подобно типу const T *
X::difference_type	Целочисленный тип со знаком, используемый для представления расстояния от одного итератора до другого (например, различие между двумя указателями)
X::size_type	Целочисленный тип size_type без знака, который может представлять размеры объектов данных, количество элементов, а также индексы

Для определения этих членов в классе используется typedef. Эти типы можно применять для объявления подходящих переменных. Например, в следующем фрагменте кода реализован обходной путь замены первого вхождения "bonus" словом "bogus" в векторе объектов string, чтобы показать, как использовать типы-члены для объявления переменных:

```
using namespace std;
vector<string> input;
string temp;
while (cin >> temp && temp != "quit")
   input.push_back(temp);
vector<string>::iterator want=
   find(input.begin(), input.end(), string("bonus"));
if (want != input.end())
{
   vector<string>::reference r = *want;
   r = "bogus";
}
```

В этом коде r становится ссылкой на элемент в input, на который указывает want. Аналогично, продолжая предыдущий пример, можно написать следующий код:

```
vector<string>::value_type s1 = input[0];    // s1 имеет тип string
vector<string>::reference s2 = input[1];     // s2 имеет тип string &
```

В результате s1 становится новым объектом string, который является копией input[0], а s2 — ссылкой на input[1]. Этот пример можно упростить, если уже известно, что шаблон основан на типе string:

```
string s1 = input[0];                        // s1 имеет тип string
string & s2 = input[1];                      // s2 имеет тип string &
```

Однако более сложные типы из табл. Ж.1 можно также использовать в более общем коде, в котором тип контейнера и элемента является обобщенным. Предположим, например, что требуется функция min(), которая принимает ссылку на контейнер в качестве своего аргумента и возвращает наименьший элемент в контейнере. Здесь предполагается, что для типа значения, применяемого для реализации шаблона, определена операция <, и вы не хотите использовать алгоритм min_element() из STL, который работает с интерфейсом итераторов. Поскольку в качестве аргумента может выступать vector<int>, list<string> или deque<double>, то для представления этого контейнера вы используете шаблон с шаблонным параметром, таким как Bag. (Другими словами, Bag является шаблонным типом, который можно реализовать как vector<int>, list<string> или другой контейнерный тип.) Таким образом, типом аргумента для функции является const Bag & b. А что можно сказать о возвращаемом типе? Это должен быть тип значения для контейнера, т.е. Bag::value_type. Однако на данный момент Bag является просто шаблонным параметром и компилятор не имеет возможности узнать о том, что член value_type на самом деле является типом. С помощью ключевого слова typename можно уточнить, что член класса — это typedef:

```
vector<string>::value_type st;     // vector<string> - определенный класс
typename Bag::value_type m;        // Bag — пока еще не определенный тип
```

Здесь в первом определении компилятор получает доступ к определению шаблона vector, в котором говорится, что value_type — это typedef. Во втором определении ключевое слово typename гарантирует, что каким бы ни был Bag, комбинация Bag::value_type является именем типа. Эти соображения приводят к следующему определению:

```
template<typename Bag>
typename Bag::value_type min(const Bag & b)
{
   typename Bag::const_iterator it;
   typename Bag::value_type m = *b.begin();
   for (it = b.begin(); it != b.end(); ++it)
      if (*it < m)
   m = *it;
   return m;
}
```

Эту шаблонную функцию впоследствии можно использовать так, как показано ниже:

```
vector<int> temperatures;
// Ввод значений температуры в вектор
int coldest = min(temperatures);
```

Параметр `temperatures` может привести к тому, что `Bag` будет интерпретироваться как `vector<int>`, а `typename Bag::value_type` — как `vector<int>::value_type`, который, в свою очередь, имеет тип `int`.

Все контейнеры также содержат функции-члены или операции, перечисленные в табл. Ж.2. В этой таблице `X` — тип контейнера, например, `vector<int>`, а `T` — тип, хранящийся в контейнере, такой как `int`. Кроме того, `a` и `b` — это значения типа `X`, `u` — идентификатор, `r` — неконстантное значение типа `X`, а `rv` — неконстантное значение rvalue типа `X`. Операции переноса были добавлены в C++11.

Таблица Ж.2. Операции, определенные для всех контейнеров

Операция	Описание
`X u;`	Конструирует пустой объект `u`
`X()`	Конструирует пустой объект
`X(a)`	Конструирует копию объекта `a`
`X u(a)`	`u` — это копия `a` (конструктор копирования)
`X u = a;`	`u` — это копия `a` (конструктор копирования)
`r = a`	`r` эквивалентно значению `a` (присваивание с копированием)
`X u(rv)`	`u` эквивалентно значению, которое было в `rv` перед конструированием (конструктор переноса)
`X u = rv;`	`u` эквивалентно значению, которое было в `rv` перед конструированием (конструктор переноса)
`a = rv`	`a` эквивалентно значению, которое было в `rv` перед конструированием (присваивание с переносом)
`(&a)->~X()`	Деструктор, применяемый к каждому элементу `a`
`begin()`	Возвращает итератор, указывающий на первый элемент
`end()`	Возвращает итератор, указывающий на элемент, следующий за последним элементом контейнера
`cbegin()`	Возвращает обратный итератор, указывающий на элемент, следующий за последним элементом контейнера
`cend()`	Возвращает обратный итератор, указывающий на первый элемент
`size()`	Возвращает количество элементов
`maxsize()`	Возвращает размер наибольшего возможного контейнера
`empty()`	Возвращает `true`, если контейнер пуст
`swap()`	Обмен содержимым двух контейнеров
`==`	Возвращает `true`, если два контейнера имеют один и тот же размер и одинаковую упорядоченность элементов
`!=`	`a != b` возвращает `!(a == b)`

Контейнеры, которые используют двунаправленный итератор либо итератор с произвольным доступом (`vector`, `list`, `deque`, `queue`, `array`, `set` и `map`), являются обратимыми и предоставляют методы, перечисленные в табл. Ж.3.

Таблица Ж.3. Типы и операции, определенные для обратимых контейнеров

Операция	Описание
`X::reverse_iterator`	Обратный итератор, указывающий на тип `T`
`X::const_reverse_iterator`	Обратный итератор `const`, указывающий на тип `T`
`a.rbegin()`	Возвращает обратный итератор, указывающий на элемент, следующий за концом `a`
`a.rend()`	Возвращает обратный итератор, указывающий на начало `a`
`a.crbegin()`	Возвращает обратный итератор `const`, указывающий на элемент, следующий за концом `a`
`a.crend()`	Возвращает обратный итератор `const`, указывающий на начало `a`

Контейнеры типа неупорядоченного набора и неупорядоченной карты не обязательно должны поддерживать дополнительные операции, перечисленные в табл. Ж.4, но остальные контейнеры их поддерживают.

Таблица Ж.4. Дополнительные операции контейнеров

Операция	Описание
`<`	`a < b` возвращает `true`, если `a` лексикографически предшествует `b`
`>`	`a > b` возвращает `b < a`
`<=`	`a <= b` возвращает `!(a > b)`
`>=`	`a >= b` возвращает `!(a < b)`

Операция > для контейнера предполагает, что для типа значения определена операция >. Лексикографическое сравнение представляет собой обобщенную версию алфавитной сортировки. При этом два контейнера сравниваются поэлементно до тех пор, пока не будет обнаружен элемент одного контейнера, который не равен соответствующему элементу другого контейнера. В этом случае считается, что контейнеры имеют тот же порядок, как и у несоответствующей пары элементов. Например, если два контейнера идентичны по первым десяти элементам, но одиннадцатый элемент первого контейнера меньше одиннадцатого элемента второго контейнера, то первый контейнер предшествует второму. Если два контейнера имеют разную длину, и элементы более короткого контейнера эквивалентны соответствующим элементам другого контейнера, то контейнер с меньшей длиной будет предшествовать контейнеру с большей длиной.

Дополнительные члены для контейнеров последовательностей

Шаблонные классы `vector`, `forward_list`, `list`, `deque` и `array` представляют собой контейнеры последовательностей, и все они имеют ранее перечисленные методы, за исключением того, что `forward_list` не является обращаемым, поэтому он не поддерживает методы из табл. Ж.3. Контейнер последовательности хранит однородный набор элементов в линейном порядке. Если последовательность содержит

фиксированное количество элементов, обычным выбором является array. С другой стороны, класс vector, который комбинирует произвольный доступ array с возможностью добавления и удаления элементов, должен быть главным ресурсом. Однако если требуются частые добавления в середину последовательности, подумайте об использовании list или forward_list. Если же добавления и удаления в основном случаются с двух концов последовательности, выбирайте deque.

Фиксированный размер объекта array не позволяет использовать многие методы последовательностей. В табл. Ж.5 перечислены дополнительные методы, которые доступны для контейнеров последовательностей, отличных от array. (Класс forward_list имеет несколько отличающиеся определения для resize().)

Здесь: X — тип контейнера, такой как vector<int>; T — тип элементов, хранящихся в контейнере, вроде int; a — значение типа X; t — значение lvalue или const-значение rvalue типа X::value_type; rv — отличное от const значение rvalue того же самого типа; i и j — входные итераторы; [i, j) — допустимый диапазон; il — объект типа initializer_list<value_type>; p — допустимый итератор const для a; q — допустимый разыменуемый итератор const для a; [q1, q2) — допустимый диапазон для итераторов const; n — целое число типа X::size_type; Args — пакет параметров шаблона; args — пакет параметров функции вида Args&&.

Таблица Ж.5. Дополнительные операции, определенные для контейнеров последовательностей

Операция	Описание
`X(n, t)`	Конструирует контейнер последовательности с n копиями t
`X a(n, t)`	Конструирует контейнер последовательности a с n копиями t
`X(i, j)`	Конструирует контейнер последовательности, соответствующий диапазону [i, j)
`X a(i, j)`	Конструирует контейнер последовательности a, соответствующий диапазону [i, j)
`X(il);`	Конструирует контейнер последовательности, инициализированный содержимым il
`a = il;`	Копирует значения из il в a
`a.emplace(p, args);`	Вставляет объект типа T перед p, используя конструктор T с аргументами, которые упакованы в args
`a.insert(p,t)`	Вставляет копию t перед p; возвращает итератор, указывающий на вставленную копию t. Значением по умолчанию для t является T(), т.е. значение, используемое для типа T при отсутствии явной инициализации
`a.insert(p,rv)`	Вставляет копию rv перед p; возвращает итератор, указывающий на вставленную копию rv. Может использоваться семантика переноса
`a.insert(p,n,t)`	Вставляет n копий t перед p
`a.insert(p,i,j)`	Вставляет копии элементов из диапазона [i, j) перед p
`a.insert(p,il)`	То же самое, что и `a.insert(p, il.begin(), il.end())`
`a.resize(n)`	Если n > a.size(), то вставляет n – a.size() элементов перед a.end(); значением для новых элементов является значение, используемое для типа T при отсутствии явной инициализации. Если n < a.size(), то элементы, следующие за n-м элементом, очищаются

Окончание табл. Ж.5

Операция	Описание
`a.resize(n,t)`	Если `n > a.size()`, то вставляет `n - a.size()` копий `t` перед `a.end()`. Если `n < a.size()`, то элементы, следующие за `n`-м элементом, очищаются
`a.assign(i,j)`	Заменяет текущее содержимое `a` копиями элементов из диапазона [`i`, `j`)
`a.assign(n,t)`	Заменяет текущее содержимое `a` `n` копиями `t`. Значением по умолчанию для `t` является `T()`, т.е. значение, используемое для типа `T` при отсутствии явной инициализации
`a.assign(il)`	То же самое, что и `a.assign(il.begin(), il.end())`
`a.erase(q)`	Очищает элемент, на который указывает `q`; возвращает итератор на элемент, который следует за `q`
`a.erase(q1,q2)`	Очищает элементы из диапазона [`q1`, `q2`); возвращает итератор, указывающий на элемент, на который изначально указывал `q2`
`a.clear()`	Выполняет то же самое, что и `erase(a.begin(), a.end())`
`a.front()`	Возвращает `*a.begin()` (первый элемент)

В табл. Ж.6 описаны методы, общие для некоторых классов последовательностей (`vector`, `forward_list`, `list` и `deque`).

Таблица Ж.6. Операции, определенные для некоторых последовательностей

Операция	Описание	Контейнер
`a.back()`	Возвращает `*a.end()` (последний элемент)	`vector`, `list`, `deque`
`a.push_back(t)`	Вставляет `t` перед `a.end()`	`vector`, `list`, `deque`
`a.push_back(rv)`	Вставляет `t` перед `a.end()`. Может Эиспользоваться семантика переноса	`vector`, `list`, `deque`
`a.pop_back()`	Очищает последний элемент	`vector`, `list`, `deque`
`a.emplace_back()`	Добавляет объект типа `T`, используя конструктор `T` с аргументами, которые упакованы в `args`	`vector`, `list`, `deque`
`a.push_front(t)`	Вставляет копию `t` перед первым элементом	`forward_list`, `list`, `deque`
`a.push_front(rv)`	Вставляет копию `rv` перед первым элементом. Может использоваться семантика переноса	`forward_list`, `list`, `deque`
`a.emplace_front()`	Вставляет в начале объект типа `T`, используя конструктор `T` с аргументами, которые упакованы в `args`	`forward_list`, `list`, `deque`
`a.pop_front()`	Очищает первый элемент	`forward_list`, `list`
`a[n]`	Возвращает `*(a.begin() + n)`	`vector`, `deque`, `array`
`a.at(n)`	Возвращает `*(a.begin() + n)`; генерирует исключение `out_of_range`, если `n > a.size()`	`vector`, `deque`, `array`

Шаблон `vector` дополнительно имеет методы, перечисленные в табл. Ж.7. Здесь `a` — контейнер `vector`, а `n` — целое число типа `X::size_type`.

Таблица Ж.7. Дополнительные операции для векторов

Операция	Описание
`a.capacity()`	Возвращает общее количество элементов, которые может вмещать вектор, не требуя повторного размещения
`a.reserve(n)`	Сигнализирует объекту `a` о том, что необходима память как минимум для `n` элементов. После вызова метода вместимость будет составлять, по меньшей мере, `n` элементов. Повторное размещение происходит в тех случаях, когда `n` больше текущей вместимости. Если `n` больше чем `a.max_size()`, метод генерирует исключение `length_error`

Шаблон `list` дополнительно имеет методы, перечисленные в табл. Ж.8. Здесь: `a` и `b` — контейнеры `list`; `T` — тип, хранящийся в списке, такой как `int`; `t` — значение типа `T`; `i` и `j` — входные итераторы; `q2` и `p` — итераторы; `q` и `q1` — разыменуемые итераторы; `n` — целое число типа `X::size_type`. В таблице используется принятая в STL нотация `[i, j)`, которая обозначает диапазон от `i` до `j`, включая `i` и не включая `j`.

Таблица Ж.8. Дополнительные операции для списков

Операция	Описание
`a.splice(p,b)`	Перемещает содержимое списка `b` в список `a`, вставляя его перед `p`
`a.splice(p,b,i)`	Перемещает элемент, указанный `i`, из списка `b` в список `a` перед позицией `p`
`a.splice(p,b,i,j)`	Перемещает элементы списка `b` из диапазона `[i, j)` в список `a` перед позицией `p`
`a.remove(const T& t)`	Удаляет из списка `a` все элементы, которые имеют значение `t`
`a.remove_if(Predicate pred)`	При условии, что `i` является итератором для списка `a`, очищает все значения, для которых `pred(*i)` равно `true`. (`Predicate` — это булевская функция или функциональный объект, которые рассматриваются в главе 15.)
`a.unique()`	Очищает все элементы, кроме первого, из каждой группы последовательных эквивалентных элементов
`a.unique(BinaryPredicate bin_pred)`	Очищает все элементы, кроме первого, из каждой группы последовательных элементов, для которых `bin_pred(*i, *(i - 1))` равно `true`. (`BinaryPredicate` — это булевская функция или функциональный объект, которые рассматриваются в главе 15.)
`a.merge(b)`	Объединяет содержимое списка `b` с содержимым списка `a`, используя операцию `<`, которая определена для типа значения. Если элемент в списке `a` эквивалентен элементу в списке `b`, то элемент списка `a` помещается первым. После объединения список `b` становится пустым
`a.merge(b, Compare comp)`	Объединяет содержимое списка `b` с содержимым списка `a`, используя функцию `comp` или функциональный объект. Если элемент в списке `a` эквивалентен элементу в списке `b`, то элемент списка `a` помещается первым. После объединения список `b` становится пустым

Окончание табл. Ж.8

Операция	Описание
`a.sort()`	Сортирует список `a` с использованием операции <
`a.sort(Compare comp)`	Сортирует список `a`, используя функцию или функциональный объект `comp`
`a.reverse()`	Изменяет порядок элементов в списке `a` на противоположный

Операции `forward-list` аналогичны. Однако поскольку итератор шаблонного класса `forward_list` не может перемещаться назад, некоторые методы должны быть уточнены. Соответственно, методы `insert()`, `erase()` и `splice()` заменены методами `insert_after()`, `erase_after()` и `splice_after()`, имеющими дело с позицией, которая находится после позиции итератора, а не перед ней.

Дополнительные операции для множеств и карт

Ассоциативные контейнеры, моделями которых являются карты и множества, имеют шаблонные параметры `Key` и `Compare`, которые определяют, соответственно, тип ключа, используемого для упорядочения содержимого, и функциональный объект, называемый *объектом сравнения*, который применяется для сравнения значений ключа. Контейнеры `set` и `multiset` хранят ключи в виде значений, поэтому ключ имеет тот же тип, что и значение. В контейнерах `map` и `multimap` сохраненные значения одного типа (шаблонный параметр `T`) связаны с типом ключа (шаблонный параметр `Key`), а типом значения является `pair<const Key, T>`. Ассоциативные контейнеры имеют дополнительные члены для описания этих особенностей (табл. Ж.9).

Таблица Ж.9. Типы, определенные для ассоциативных контейнеров

Тип	Значение
`X::key_type`	Параметр `Key`, тип ключа
`X::key_compare`	Параметр `Compare`, который имеет значение по умолчанию `less<key_type>`
`X::value_compare`	Тип бинарного предиката, такой же, как и `key_compare` для `set` и `multuset`; он определяет порядок значений `pair<const Key, T>` в контейнере `map` или `multimap`
`X::mapped_type`	Параметр `T`, тип ассоциативных данных (только `map` и `multimap`)

Ассоциативные контейнеры предоставляют методы, перечисленные в табл. Ж.10. В общем случае объект сравнения не требует, чтобы значения с одним и тем же ключом были идентичными друг другу; термин *эквивалентные ключи* означает, что два значения, которые могут или не могут быть идентичными, имеют один и то же ключ. В этой таблице `X` — класс контейнера, а `a` — объект типа `X`. Если `X` использует несколько ключей (т.е. является `multiset` или `multimap`), то `a_eq` представляет собой объект типа `X`. Как обычно, `i` и `j` — входные итераторы, ссылающиеся на элементы `value_type`, `[i, j)` — допустимый диапазон, `p` и `q2` — итераторы по `a`, `q` и `q1` — разыменуемые итераторы по `a`, `[q1, q2)` — допустимый диапазон, `t` — значение `X::value_type` (которое может быть парой значений), а `k` — значение `X::key_type`. Наконец, `il` — это объект `initializer_list<value_type>`.

Таблица Ж.10. Операции, определенные для множеств, мультимножеств, карт и мультикарт

Операция	Описание
`X(i,j,c)`	Конструирует пустой контейнер, вставляет элементы из диапазона [i, j) и применяет c в качестве объекта сравнения
`X a(i,j,c)`	Конструирует пустой контейнер (a), вставляет элементы из диапазона [i, j) и применяет c в качестве объекта сравнения
`X(i,j)`	Конструирует пустой контейнер, вставляет элементы из диапазона [i, j) и применяет `Compare()` в качестве объекта сравнения
`X a(i,j)`	Конструирует пустой контейнер (a), вставляет элементы из диапазона [i, j) и применяет `Compare()` в качестве объекта сравнения
`X(il);`	То же самое, что и `X(il.begin(), il.end())`
`a = il`	Присваивает a диапазон `il.begin()`, `il.end()`
`a.key_comp()`	Возвращает объект сравнения, используемый при создании a
`a.value_comp()`	Возвращает объект типа `value_compare_type`
`a_uniq.insert(t)`	Вставляет значение t в контейнер a, если и только если a еще не содержит значения с эквивалентным ключом. Метод возвращает значение типа `pair<iterator,bool>`. Компонент `bool` равен `true`, если вставка была осуществлена, и `false` — в противном случае. Компонент итератора указывает на элемент, ключ которого эквивалентен ключу t
`a_eq.insert(t)`	Вставляет t и возвращает итератор, указывающий на эту позицию
`a.insert(p,t)`	Вставляет t, используя p в качестве подсказки, где функция `insert()` должна начинать поиск. Если a является контейнером с уникальными ключами, то вставка будет произведена только в том случае, если a не будет содержать элемент с эквивалентным ключом; в противном случае вставка не выполняется. Вне зависимости от того, производится вставка или нет, метод возвращает итератор, указывающий на позицию с эквивалентным ключом
`a.insert(i,j)`	Вставляет в a элементы из диапазона [i, j)
`a.insert(il)`	Вставляет в a элементы из il типа `initializer_list`
`a_uniq.emplace(args)`	Подобна `a_uniq.insert(t)`, но использует конструктор T, список параметров которого соответствует содержимому пакета параметров `args`
`a_eq.emplace(args)`	Подобна `a_eq.insert(t)`, но использует конструктор T, список параметров которого соответствует содержимому пакета параметров `args`
`a.emplace_hint(args)`	Подобна `a.insert(p,t)`, но использует конструктор T, список параметров которого соответствует содержимому пакета параметров `args`
`a.erase(k)`	Очищает все элементы в a, ключи которых эквивалентны k, и возвращает количество очищенных элементов
`a.erase(q)`	Очищает элемент, на который указывает q
`a.erase(q1,q2)`	Очищает элементы из диапазона [q1, q2)
`a.clear()`	Делает то же самое, что и `erase(a.begin(), a.end())`
`a.find(k)`	Возвращает итератор, указывающий на элемент, ключ которого эквивалентен k; если такой элемент не найден, возвращает `a.end()`

Окончание табл. Ж.10

Операция	Описание
`a.count(k)`	Возвращает количество элементов, которые имеют ключи, эквивалентные `k`
`a.lower_bound(k)`	Возвращает итератор на первый элемент, ключ которого не меньше `k`
`a.upper_bound(k)`	Возвращает итератор на первый элемент, ключ которого больше `k`
`a.equal_range(k)`	Возвращает пару, первым членом которой является `a.lower_bound(k)`, а вторым — `a.upper_bound(k)`
`a.operator[](k)`	Возвращает ссылку на значение, связанное с ключом `k` (только для контейнеров `map`)

Неупорядоченные ассоциативные контейнеры (C++11)

Как упоминалось ранее, неупорядоченные ассоциативные контейнеры (`unordered_set`, `unordered_multiset`, `unordered_map` и `unordered_multimap`) используют ключи и хеш-таблицы для предоставления быстрого доступа к данным. Давайте кратко рассмотрим эти концепции. *Хеш-функция* преобразует ключ в значение индекса. Например, если ключ имеет тип `string`, хеш-функция могла бы суммировать числовые коды символов в `string` и вычислять эту сумму по модулю 13, давая в результате индексы в диапазоне 0–12. Неупорядоченный контейнер будет использовать для хранения объектов `string` 13 *областей*. Объект `string`, скажем, с индексом 4, будет помещен в область 4. Чтобы найти контейнер для ключа, необходимо применить хеш-функцию к ключу и просто искать область с соответствующим индексом. В идеальном случае областей должно быть достаточно для того, чтобы каждая из них содержала лишь несколько объектов `string`.

Библиотека C++11 предоставляет шаблон `hash<Key>`, который неупорядоченные ассоциативные контейнеры используют по умолчанию. Также определены специализации для разнообразных целочисленных типов и типов с плавающей точкой, указателей и ряда шаблонных классов, таких как `string`.

Типы, используемые для этих контейнеров, перечислены в табл. Ж.11.

Интерфейс для неупорядоченных ассоциативных контейнеров похож на интерфейс обычных ассоциативных контейнеров. В частности, операции из табл. Ж.10 также применимы к неупорядоченным ассоциативным контейнерам со следующим исключением: методы `lower_bound()` и `upper_bound()` не требуются, равно как и конструктор `X(i,j,c)`. Тот факт, что обычные ассоциативные контейнеры являются упорядоченными, позволяет им пользоваться предикатом сравнения, который выражает концепцию "меньше чем". Такое сравнение неприменимо к неупорядоченным ассоциативным контейнерам, поэтому они вместо него используют предикат сравнения, основанный на концепции "является эквивалентным".

В дополнение к методам из табл. Ж.10, неупорядоченные ассоциативные контейнеры требуют ряда дополнительных методов, которые перечислены в табл. Ж.12.

В этой таблице `X` — класс неупорядоченного ассоциативного контейнера, `a` — объект типа `X`, `b` — возможно константный объект типа `X`, `a_uniq` — объект типа `unordered_set` или `unordered_map`, `a_eq is` — объект типа `unordered_multiset` или `unordered_multimap`, `hf` — значение типа `hasher`, `eq` — значение типа `key_equal`, `n` — значение типа `size_type`, а `z` — значение типа `float`.

Таблица Ж.11. Типы, определенные для неупорядоченных ассоциативных контейнеров

Тип	Значение
`X::key_type`	Параметр `Key`, тип ключа
`X::key_equal`	Параметр `Pred`, представляющий собой бинарный предикат, который проверяет, эквивалентны ли два аргумента типа `Key`
`X::hasher`	Параметр `Hash`, тип унарного функционального объекта, такой, что если `hf` имеет тип `Hash` и `k` имеет тип `Key`, то `hf(k)` имеет тип `std::size_t`
`X::local_iterator`	Итератор того же типа, что и `X::iterator`, но который может использоваться только внутри одиночной области
`X::const_local_iterator`	Итератор того же типа, что и `X::const_iterator`, но который может использоваться только внутри одиночной области
`X::mapped_type`	`T`, тип связанных данных (только `map` и `multimap`)

Как и ранее, `i` и `j` — входные итераторы, ссылающиеся на элементы `value_type`, `[i, j)` — допустимый диапазон, `p` и `q2` — итераторы для `a`, `q` и `q1` — разыменуемые итераторы для `a`, `[q1, q2)` — допустимый диапазон, `t` — значение типа `X::value_type` (которое может быть парой) и `k` — значение типа `X::key_type`. Кроме того, `il` — это объект `initializer_list<value_type>`.

Таблица Ж.12. Дополнительные операции, определенные для неупорядоченных множеств, мультимножеств, карт и мультикарт

Операция	Описание
`X(n,hf,eq)`	Конструирует пустой контейнер с как минимум `n` областями, используя `hf` в качестве хеш-функции и `eq` в качестве предиката эквивалентности ключей. Если параметр `eq` опущен, для предиката эквивалентности ключей применяется `key_equal()`.Если параметр `hf` также опущен, для хеш-функции используется `hasher()`
`X a(n,hf,eq)`	Конструирует пустой контейнер `a` с как минимум `n` областями, используя `hf` в качестве хеш-функции и `eq` в качестве предиката эквивалентности ключей. Если параметр `eq` опущен, для предиката эквивалентности ключей применяется `key_equal()`.Если параметр `hf` также опущен, для хеш-функции используется `hasher()`
`X(i,j,n,hf,eq)`	Конструирует пустой контейнер с как минимум `n` областями, используя `hf` в качестве хеш-функции и `eq` в качестве предиката эквивалентности ключей, и вставляет элементы из `[i, j)`. Если параметр `eq` опущен, для предиката эквивалентности ключей применяется `key_equal()`. Если параметр `hf` также опущен, для хеш-функции используется `hasher()`. Если параметр `n` опущен, применяется неуказанное количество областей
`X a(i,j,n,hf,eq)`	Конструирует пустой контейнер `a` с как минимум `n` областями, используя `hf` в качестве хеш-функции и `eq` в качестве предиката эквивалентности ключей, и вставляет элементы из `[i, j)`. Если параметр `eq` опущен, для предиката эквивалентности ключей применяется `key_equal()`. Если параметр `hf` также опущен, для хеш-функции используется `hasher()`. Если параметр `n` опущен, применяется неуказанное количество областей
`b.hash_function()`	Возвращает хеш-функцию, используемую при конструировании `b`

Окончание табл. Ж.12

Операция	Описание
`b.key_eq()`	Возвращает предикат эквивалентности ключей, используемый при конструировании `b`
`b.bucket_count()`	Возвращает количество областей в `b`
`b.max_bucket_count()`	Возвращает верхнюю границу для количества областей, которые может содержать `b`
`b.bucket(k)`	Возвращает индекс области, которая будет содержать элемент с ключом, эквивалентным `k`
`b.bucket_size(n)`	Возвращает количество элементов в области, имеющей индекс `n`
`b.begin(n)`	Возвращает итератор для первого элемента в области, имеющей индекс `n`
`b.end(n)`	Возвращает итератор для элемента, находящегося за последним, в области, имеющей индекс `n`
`b.cbegin(n)`	Возвращает константный итератор для первого элемента в области, имеющей индекс `n`
`b.cbegin(n)`	Возвращает константный итератор для элемента, находящегося за последним, в области, имеющей индекс `n`
`b.load_factor()`	Возвращает среднее количество элементов на область
`b.max_load_factor()`	Возвращает значение, используемое как максимальный коэффициент загрузки; когда коэффициент загрузки превышает это значение, контейнер увеличивает количество областей
`b.max_load_factor(z)`	Может изменить максимальный коэффициент загрузки, используя `z` в качестве подсказки
`a.rehash(n)`	Сбрасывает счетчик областей в значение >= `n` со следующим требованием: `a.bucket_count() > a.size()/a.max_load.factor()`
`a.reserve(n)`	То же самое, что и `a.rehash(ceil(n/a.max_load_factor()))`, где `ceil(x)` — это наименьшее целое значение >= `x`

Функции библиотеки STL

Библиотека алгоритмов STL, которая поддерживается заголовочными файлами `algorithm` и `numeric`, предлагает большое количество шаблонных функций, не являющихся членами, на основе итераторов. Как было сказано в главе 16, имена шаблонных параметров подбираются так, чтобы отражать моделируемые ими концепции. Например, имя `ForwardIterator` показывает, что параметр должен, как минимум, моделировать требования однонаправленного итератора, а имя `Predicate` используется, чтобы отразить параметр, который должен быть функциональным объектом с одним аргументом и возвращаемым значением `bool`. Стандарт C++ разделяет алгоритмы на четыре группы: операции, не модифицирующие последовательности; операции, видоизменяющие последовательности; операции сортировки и связанные с ними операции; числовые операции. (В C++11 числовые операции перемещены из STL в заголовочный файл `numeric`, но это никак не влияет на способы их применения.) Термин *последовательная операция* говорит о том, что функция принимает пару итераторов в качестве аргументов, чтобы определить диапазон, или последовательность, с которой будут производиться действия. Термин *видоизменяющий* означает, что функция может изменять контейнер.

Операции, не модифицирующие последовательности

В табл. Ж.13 перечислены операции, не изменяющие последовательности. Аргументы в этой таблице не показаны, а перегруженные функции представлены только один раз. Более подробное их описание, включая прототипы, приведено после таблицы. Итак, вы можете просмотреть таблицу, чтобы узнать, что выполняет определенная функция, и затем ознакомиться с детальной информацией о ней.

Таблица Ж.13. Операции, не изменяющие последовательности

Функция	Описание
`all_of()`	Возвращает `true`, если проверка предиката дает `true` для всех элементов (C++11)
`any_of()`	Возвращает `true`, если проверка предиката дает `true` для любого элемента (C++11)
`none_of()`	Возвращает `true`, если проверка предиката дает `false` для всех элементов (C++11)
`for_each()`	Применяет не изменяющий функциональный объект к каждому элементу диапазона
`find()`	Находит первое вхождение значения в диапазоне
`find_if()`	Находит первое значение, которое удовлетворяет предикату в диапазоне
`find_if_not()`	Находит первое значение, которое не удовлетворяет предикату в диапазоне (C++11)
`find_end()`	Находит последнее вхождение подпоследовательности, значения которой совпадают со значениями второй последовательности. Совпадение может определяться равенством или бинарным предикатом
`find_first_of()`	Находит первое вхождение любого элемента второй последовательности, который совпадает со значением в первой последовательности. Совпадение может определяться равенством или бинарным предикатом
`adjacent_find()`	Находит первый элемент, который совпадает с соседним элементом. Совпадение может определяться равенством или бинарным предикатом
`count()`	Возвращает количество обнаружений данного значения в диапазоне
`count_if()`	Возвращает количество обнаружений данного значения в диапазоне, при этом совпадение определяется бинарным предикатом
`mismatch()`	Находит первый элемент в одном диапазоне, который не совпадает с соответствующим элементом во втором диапазоне, и возвращает итераторы для каждого из них. Совпадение определяться равенством или бинарным предикатом
`equal()`	Возвращает `true`, если каждый элемент в одном диапазоне совпадает с соответствующим элементом во втором диапазоне. Совпадение может определяться равенством или бинарным предикатом
`is_permutation()`	Возвращает `true`, если каждый элемент в одном диапазоне совпадает с соответствующим элементом в некоторой перестановке второго диапазона. Совпадение может определяться равенством или бинарным предикатом (C++11)
`search()`	Находит первое вхождение подпоследовательности, значения которой совпадают со значениями второй последовательности. Совпадение может определяться равенством или бинарным предикатом
`search_n()`	Находит первую подпоследовательность из `n` элементов, которые совпадают с данным значением. Совпадение может определяться равенством или бинарным предикатом

Теперь давайте рассмотрим более подробно операции, не изменяющие последовательности. Для каждой функции показан прототип (прототипы), за которым следует краткое пояснение. Пары итераторов отражают диапазоны, с выбранным именем шаблонного параметра, указывающим на тип итератора. Как обычно, диапазон определяется в виде [first, last), включая first и не включая last. Некоторые функции принимают два диапазона, которые не могут находиться в одном и том же контейнере. Например, equal() можно применять для сравнения списка с вектором.

Функции, передаваемые в виде аргументов, являются функциональными объектами, которые могут быть указателями (примером могут быть имена функций) или объектами, имеющими определенную операцию (). Как и в главе 16, предикат представляет собой булевскую функцию с одним аргументом, а бинарным предикатом является булевская функция с двумя аргументами. (Функции не обязательно должны быть типа bool, поскольку они возвращают значение 0, соответствующее значению false, и ненулевое значение, соответствующее значению true.)

all_of() (C++11)

```
template<class InputIterator, class Predicate>
bool all_of(InputIterator first, InputIterator last, Predicate pred);
```

Функция all_of() возвращает true, если pred(*i) равно true для каждого итератора в диапазоне [first, last), или в случае пустого диапазона. Иначе эта функция возвращает false.

any_of() (C++11)

```
template<class InputIterator, class Predicate>
bool any_of(InputIterator first, InputIterator last, Predicate pred);
```

Функция any_of() возвращает false, если pred(*i) равно false для каждого итератора в диапазоне [first, last), или в случае пустого диапазона. Иначе эта функция возвращает true.

none_of() (C++11)

```
template<class InputIterator, class Predicate>
bool none_of(InputIterator first, InputIterator last, Predicate pred);
```

Функция none_of() возвращает true, если pred(*i) равно false для каждого итератора в диапазоне [first, last), или в случае пустого диапазона. Иначе эта функция возвращает false.

for_each()

```
template<class InputIterator, class Function>
Function for_each(InputIterator first, InputIterator last, Function f);
```

Функция for_each() применяет функциональный объект f к каждому элементу в диапазоне [first, last). Она также возвращает f.

find()

```
template<class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& value);
```

Функция find() возвращает итератор на первый элемент в диапазоне [first, last), который имеет значение value; если элемент не найден, она возвращает last.

ind_if()

```
template<class InputIterator, class Predicate>
InputIterator find_if(InputIterator first, InputIterator last, Predicate pred);
```

Функция `find_if()` возвращает итератор `it` на первый элемент в диапазоне [`first`, `last`), для которого вызов функционального объекта `pred(*i)` дает `true`; если элемент не найден, функция возвращает `last`.

find_if_not()

```
template<class InputIterator, class Predicate>
InputIterator find_if_not(InputIterator first, InputIterator last,
                          Predicate pred);
```

Функция `find_if_not()` возвращает итератор `it` для первого элемента в диапазоне [`first`, `last`), для которого вызов функционального объекта `pred(*i)` дает `false`; если элемент не найден, функция возвращает `last`.

find_end()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_end(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2, class
BinaryPredicate>
ForwardIterator1 find_end(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2, BinaryPredicate pred);
```

Функция `find_end()` возвращает итератор `it` для последнего элемента в диапазоне [`first1`, `last1`), который отмечает начало подпоследовательности, совпадающей с содержимым диапазона [`first2`, `last2`). В первой версии при сопоставлении элементов используется операция ==, определенная для конкретного типа значения. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката `pred`. Таким образом, элементы, на которые указывает `it1` и `it2`, совпадают, если результат `pred(*it1, *it2)` равен `true`. Обе версии возвращают `last1`, если элемент не найден.

find_first_of()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 find_first_of(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
class BinaryPredicate>
ForwardIterator1 find_first_of(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2,
   BinaryPredicate pred);
```

Функция `find_first()` возвращает итератор `it` для первого элемента в диапазоне [`first1`, `last1`), который совпадает с любым элементом в диапазоне [`first2`, `last2`). В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется функциональный

объект бинарного предиката pred. Таким образом, элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true. Обе версии возвращают last1, если элемент не найден.

adjacent_find()

```
template<class ForwardIterator>
ForwardIterator adjacent_find(ForwardIterator first,
                              ForwardIterator last);
template<class ForwardIterator, class BinaryPredicate>
ForwardIterator adjacent_find(ForwardIterator first,
                              ForwardIterator last,
                              BinaryPredicate pred);
```

Функция adjacent_find() возвращает итератор it для первого элемента в диапазоне [first1, last1), при условии, что элемент совпадает со следующим элементом. Функция возвращает last, если такая пара элементов не найдена. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката pred. Таким образом, элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true.

count()

```
template<class InputIterator, class T>
   typename iterator_traits<InputIterator>::difference_type count(
      InputIterator first, InputIterator last, const T& value);
```

Функция count() возвращает количество элементов в диапазоне [first, last), которые совпадают со значением value. Для сопоставления элементов используется операция == для типа значения. Возвращаемым типом является целочисленный тип, способный уместить максимальное количество элементов, которые может содержать контейнер.

count_if()

```
template<class InputIterator, class Predicate>
   typename iterator_traits<InputIterator>::difference_type count_if(
      InputIterator first, InputIterator last, Predicate pred);
```

Функция count_if() возвращает количество элементов в диапазоне [first, last), для которых функциональный объект pred возвращает значение true при передаче элемента в качестве аргумента.

mismatch()

```
template<class InputIterator1, class InputIterator2>
   pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2);
template<class InputIterator1, class InputIterator2, class BinaryPredicate>
   pair<InputIterator1, InputIterator2> mismatch(InputIterator1 first1,
      InputIterator1 last1, InputIterator2 first2, BinaryPredicate pred);
```

Каждая из функций mismatch() находит первый элемент в диапазоне [first1, last1), который не совпадает с соответствующим элементом в диапазоне, начинающемся с first2, и возвращает пару, содержащую итераторы для двух несовпадающих элементов. Если не найдено ни одного несовпадения, возвращаемым значением будет

pair<last1, first2 + (last1 – first1)>. В первой версии для сопоставления элементов используется операция ==, определенная для конкретного типа значсния. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката pred. Таким образом, элементы, на которые указывают it1 и it2, не совпадают, если результат pred(*it1, *it2) равен false.

equal()

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2);
template<class InputIterator1, class InputIterator2,
       class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, BinaryPredicate pred);
```

Функция equal() возвращает значение true, если каждый элемент в диапазоне [first1, last1) совпадает с соответствующим элементом в последовательности, начинающейся с first2; в противном случае функция возвращает false. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката pred. Таким образом, элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true.

is_permutation() (C++11)

```
template<class InputIterator1, class InputIterator2>
bool equal(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2);
template<class InputIterator1, class InputIterator2,
       class BinaryPredicate>
bool equal(InputIterator1 first1, InputIterator1 last1,
        InputIterator2 first2, BinaryPredicate pred);
```

Функция is_permutation() возвращает true, если каждый элемент в диапазоне [first1, last1) совпадает с соответствующим элементом в некоторой перестановке последовательности эквивалентной длины, начинающейся с first2; в противном случае она возвращает false. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката pred. Таким образом, элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true.

search()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator1 search(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2);
template<class ForwardIterator1, class ForwardIterator2,
       class BinaryPredicate>
ForwardIterator1 search(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2, ForwardIterator2 last2,
   BinaryPredicate pred);
```

Функция search() находит первое вхождение в диапазоне [first1, last1), которое совпадает с соответствующей последовательностью, найденной в диапазоне [first2, last2). Функция возвращает last1, если такая последовательность не найдена. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката pred. Таким образом, элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true.

search_n()

```
template<class ForwardIterator, class Size, class T>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value);
template<class ForwardIterator, class Size, class T, class BinaryPredicate>
ForwardIterator search_n(ForwardIterator first, ForwardIterator last,
                         Size count, const T& value, BinaryPredicate pred);
```

Функция search_n() находит первое вхождение в диапазоне [firts1, last1), которое совпадает с последовательностью, состоящей из count последовательных вхождений значения value. Функция возвращает last1, если такая последовательность не найдена. В первой версии для сопоставления элементов используется операция == для типа значения. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката pred. Таким образом, элементы, на которые указывает it1 и it2, совпадают, если результат pred(*it1, *it2) равен true.

Операции, видоизменяющие последовательности

В табл. Ж.14 перечислены операции, видоизменяющие последовательности. Аргументы в этой таблице не показаны, а перегруженные функции представлены только один раз. За таблицей следует более подробное их описание, включая прототипы. Таким образом, вы можете просмотреть таблицу, чтобы узнать, что выполняет определенная функция, после чего обратиться к детальной информации о ней.

Таблица Ж.14. Операции, видоизменяющие последовательности

Функция	Описание
copy()	Копирует элементы из диапазона в позицию, идентифицируемую итератором
copy_n()	Копирует n элементов из одной позиции, указанной итератором, в другую позицию, идентифицируемую итератором (C++11)
copy_if()	Копирует элементы, удовлетворяющие предикату, из диапазона в позицию, идентифицируемую итератором (C++11)
copy_backward()	Копирует элементы из диапазона в позицию, идентифицируемую итератором. Копирование начинается с конца диапазона и продолжается в обратном порядке
move()	Перемещает элементы из диапазона в позицию, идентифицируемую итератором (C++11)
move_backward()	Перемещает элементы из диапазона в позицию, идентифицируемую итератором. Перемещение начинается с конца диапазона и продолжается в обратном порядке (C++11)
swap()	Меняет местами два значения, которые хранятся в позициях, указанных ссылками

Продолжение табл. Ж.14

Функция	Описание
`swap_ranges()`	Меняет местами соответствующие значения в двух диапазонах
`iter_swap()`	Меняет местами два значения, которые хранятся в позициях, указанных итераторами
`transform()`	Применяет функциональный объект к каждому элементу в диапазоне (или к каждой паре элементов в паре диапазонов) и копирует возвращаемое значение в соответствующую позицию другого диапазона
`replace()`	Заменяет каждое вхождение значения в диапазоне другим значением
`replace_if()`	Заменяет каждое вхождение значения в диапазоне другим значением, если функциональный объект предиката, примененный к исходному значению, возвращает `true`
`replace_copy()`	Копирует один диапазон в другой и заменяет каждое вхождение определенного значения другим значением
`replace_copy_if()`	Копирует один диапазон в другой и заменяет каждое значение, для которого функциональный объект предиката дает `true`, указанным значением
`fill()`	Заполняет диапазон указанным значением
`fill_n()`	Присваивает указанное значение `n` последовательным элементам
`generate()`	Присваивает каждому значению в диапазоне возвращаемое значение генератора, который представляет собой функциональный объект, не принимающий аргументов
`generate_n()`	Присваивает первым `n` значениям в диапазоне возвращаемое значение генератора, который представляет собой функциональный объект, не принимающий аргументов
`remove()`	Удаляет из диапазона все вхождения указанного значения и возвращает для результирующего диапазона итератор на элемент, следующий за последним элементом
`remove_if()`	Удаляет из диапазона все вхождения значений, для которых объект предиката возвращает `true`, и возвращает для результирующего диапазона итератор на элемент, следующий за последним элементом
`remove_copy()`	Копирует элементы из одного диапазона в другой, опуская элементы, равные определенному значению
`remove_copy_if()`	Копирует элементы из одного диапазона в другой, опуская элементы, для которых функциональный объект предиката возвращает `true`
`unique()`	Сокращает каждую последовательность из двух или более эквивалентных элементов в диапазоне до одного элемента
`unique_copy()`	Копирует элементы из одного диапазона в другой, сокращая каждую последовательность из двух или более эквивалентных элементов в диапазоне до одного элемента
`reverse()`	Изменяет порядок элементов в диапазоне на противоположный
`reverse_copy()`	Копирует диапазон в обратном порядке в другой диапазон
`rotate()`	Интерпретирует диапазон как циклическое упорядочение и циклически сдвигает элементы влево
`rotate_copy()`	Копирует один диапазон в другой в циклическом порядке

Окончание табл. Ж.14

Функция	Описание
`random_shuffle()`	Случайным образом перераспределяет элементы в диапазоне
`shuffle()`	Случайным образом перераспределяет элементы в диапазоне, используя тип функционального объекта, удовлетворяющего требованиям C++11 для генераторов случайных чисел с нормальным распределением (C++11)
`is_partitioned()`	Возвращает `true`, если диапазон секционирован заданным предикатом
`partition()`	Помещает все элементы, удовлетворяющие функциональному объекту предиката, перед всеми элементами, которые ему не удовлетворяют
`stable_partition()`	Помещает все элементы, удовлетворяющие функциональному объекту предиката, перед элементами, которые ему не удовлетворяют. Сохраняет относительный порядок значений в каждой группе
`partition_copy()`	Копирует все элементы, удовлетворяющие функциональному объекту предиката, в первый выходной диапазон, а остальные элементы — во второй выходной диапазон (C++11)
`partition_point()`	Для диапазона, секционированного заданным предикатом, возвращает итератор для первого элемента, который не удовлетворяет этому предикату

Теперь давайте рассмотрим операции, видоизменяющие последовательности, более подробно. Для каждой функции показан прототип (прототипы) и приводится краткое пояснение. Пары итераторов указывают на диапазоны, с выбранным именем шаблонного параметра, который отражает тип итератора. Как обычно, диапазон определяется в виде [`first`, `last`), включая `first` и не включая `last`. Функции, передаваемые в виде аргументов, являются функциональными объектами, которые могут быть указателями или объектами, имеющими определенную операцию `()`. Как и в главе 16, предикат представляет собой булевскую функцию с одним аргументом, а бинарным предикатом является булевская функция с двумя аргументами. (Функции не обязательно должны быть типа `bool`, поскольку они возвращают значение `0`, соответствующее значению `false`, и ненулевое значение, соответствующее значению `true`.) Как и ранее, унарный функциональный объект принимает один аргумент, а бинарный — два аргумента.

copy()

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

Функция `copy()` копирует элементы из диапазона [`first`, `last`) в диапазон [`result`, `result + (last - first)`). Функция возвращает `result + (last - first)`, т.е. итератор, указывающий на позицию, следующую за последней позицией, в которую был скопирован элемент. Функция требует, чтобы `result` не находился в диапазоне [`first`, `last`), т.е. чтобы результирующий диапазон не перекрывал исходный.

copy_n() (C++11)

```
template<class InputIterator, class Size class OutputIterator>
OutputIterator copy(InputIterator first, Size n,
                    OutputIterator result);
```

Функция `copy_n()` копирует `n` элементов, начиная с позиции `first`, в диапазон [`result`, `result + n`). Она возвращает `result + n` — т.е. итератор, указывающий на

позицию, следующую за последней позицией, в которую был скопирован элемент. Функция не требует, чтобы результирующий диапазон не перекрывал исходный.

copy_if() (C++11)

```
template<class InputIterator, class OutputIterator,
       class Predicate>
OutputIterator copy_if(InputIterator first, InputIterator last,
                       OutputIterator result, Predicate pred);
```

Функция `copy_if()` копирует элементы диапазона `[first, last)`, на которые ссылается итератор `i` и для которых `pred(*i)` равно `true`, в диапазон `[result, result + (last - first))`. Она возвращает `result + (last - first)` — т.е. итератор, указывающий на позицию, следующую за последней позицией, в которую был скопирован элемент. Функция требует, чтобы `result` не находился в диапазоне `[first, last)`, т.е. чтобы результирующий диапазон не перекрывал исходный.

copy_backward()

```
template<class BidirectionalIterator1,
       class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
       BidirectionalIterator1 last, BidirectionalIterator2 result);
```

Функция `copy_backward()` копирует элементы из диапазона `[first, last)` в диапазон `[result - (last - first), result)`. Копирование начинается с элемента в позиции `last - 1`, который копируется в позицию `result - 1`, и продолжается в обратном порядке до `first`. Функция возвращает `result - (last - first)`, т.е. итератор, указывающий на позицию, следующую за последней позицией, в которую был скопирован элемент. Функция требует, чтобы `result` не находился в диапазоне `[first, last)`. Однако поскольку копирование осуществляется в обратном порядке, допускается перекрытие результирующего и исходного диапазонов.

move() (C++11)

```
template<class InputIterator, class OutputIterator>
OutputIterator copy(InputIterator first, InputIterator last,
                    OutputIterator result);
```

Функция `move()` использует `std::move()` для перемещения элементов диапазона `[first, last)` в диапазон `[result, result + (last - first))`. Она возвращает `result + (last - first)`, т.е. итератор, указывающий на позицию, следующую за последней позицией, в которую был скопирован элемент. Функция требует, чтобы `result` не находился в диапазоне `[first, last)`, т.е. чтобы результирующий диапазон не перекрывал исходный.

move_backward() (C++11)

```
template<class BidirectionalIterator1,
       class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
       BidirectionalIterator1 last, BidirectionalIterator2 result);
```

Функция `move_backward()` использует `std::move()` для перемещения элементов диапазона `[first, last)` в диапазон `[result - (last - first), result)`. Копирование начинается с элемента `last - 1`, который копируется в позицию `result - 1`, и продолжается в обратном порядке до `first`.

Функция возвращает `result - (last - first)` — т.е. итератор, указывающий на позицию, следующую за последней позицией, в которую был скопирован элемент. Функция требует, чтобы `result` не находился в диапазоне `[first, last)`. Однако поскольку копирование осуществляется в обратном порядке, допускается перекрытие результирующего и исходного диапазонов.

swap()

```
template<class T> void swap(T& a, T& b);
```

Функция `swap()` меняет местами значения, которые хранятся в двух позициях, определяемых ссылками. (В C++11 эта функция перемещена в заголовочный файл `utility`.)

swap_ranges()

```
template<class ForwardIterator1, class ForwardIterator2>
ForwardIterator2 swap_ranges(
   ForwardIterator1 first1, ForwardIterator1 last1,
   ForwardIterator2 first2);
```

Функция `swap_ranges()` меняет местами значения из диапазона `[first1, last1)` и соответствующие значения из диапазона, начинающегося с `first2`. Два диапазона не должны перекрываться.

iter_swap()

```
template<class ForwardIterator1, class ForwardIterator2>
void iter_swap(ForwardIterator1 a, ForwardIterator2 b);
```

Функция `iter_swap()` меняет местами значения, которые хранятся в двух позициях, определяемых итераторами.

transform()

```
template<class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first, InputIterator last,
                         OutputIterator result, UnaryOperation op);
template<class InputIterator1, class InputIterator2, class OutputIterator,
       class BinaryOperation>
OutputIterator transform(InputIterator1 first1, InputIterator1 last1,
                         InputIterator2 first2, OutputIterator result,
                         BinaryOperation binary_op);
```

Первая версия функции `transform()` применяет унарный функциональный объект `op` к каждому элементу в диапазоне `[first, last)` и присваивает возвращаемое значение соответствующему элементу в диапазоне, начиная с `result`. Таким образом, `*result` устанавливается в `op(*first)` и т.д. Для результирующего диапазона функция возвращает `result + (last - first)`, т.е. значение, следующее за последним значением.

Вторая версия функции `transform()` применяет бинарный функциональный объект `op` к каждому элементу в диапазоне `[first1, last1)` и к каждому элементу в диапазоне `[first2, last2)`, после чего присваивает возвращаемое значение соответствующему элементу в диапазоне, начиная с `result`. Таким образом, `*result` устанавливается в `op(*first1, *first2)` и т.д. Для результирующего диапазона функция возвращает `result + (last - first)`, т.е. значение, следующее за последним значением.

replace()

```
template<class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last,
          const T& old_value, const T& new_value);
```

Функция replace() заменяет каждое вхождение значения old_value в диапазоне [first, last) значением new_value.

replace_if()

```
template<class ForwardIterator, class Predicate, class T>
void replace_if(ForwardIterator first, ForwardIterator last,
             Predicate pred, const T& new_value);
```

Функция replace_if() заменяет каждое значение old в диапазоне [first, last), для которого результат pred(old) равен true, значением new_value.

replace_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator replace_copy(InputIterator first, InputIterator last,
   OutputIterator result,const T& old_value, const T& new_value);
```

Функция replace_copy() копирует элементы из диапазона [first, last) в диапазон, начинающийся с result, при этом каждое значение old_value заменяется значением new_value. Для результирующего диапазона функция возвращает result + (last - first), т.е. значение, следующее за последним значением.

replace_copy_if()

```
template<class Iterator, class OutputIterator, class Predicate, class T>
OutputIterator replace_copy_if(Iterator first, Iterator last,
   OutputIterator result, Predicate pred, const T& new_value);
```

Функция replace_copy_if() копирует элементы из диапазона [first, last) в диапазон, начинающийся с result, при этом каждое значение old, для которого pred(old) равно true, заменяется значением new_value. Для результирующего диапазона функция возвращает result + (last - first), т.е. значение, следующее за последним значением.

fill()

```
template<class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& value);
```

Функция fill() присваивает каждому элементу в диапазоне [first, last) значение value.

fill_n()

```
template<class OutputIterator, class Size, class T>
void fill_n(OutputIterator first, Size n, const T& value);
```

Функция fill_n() присваивает n первым элементам, начиная с позиции first, значение value.

generate()

```
template<class ForwardIterator, class Generator>
void generate(ForwardIterator first, ForwardIterator last, Generator gen);
```

Функция generate() присваивает каждому элементу в диапазоне [first, last) значение gen(), где gen — функциональный объект генератора, т.е. функция, не принимающая аргументов. Например, gen может быть указателем на функцию rand().

generate_n()

```
template<class OutputIterator, class Size, class Generator>
void generate_n(OutputIterator first, Size n, Generator gen);
```

Функция generate_n() присваивает первым n элементам в диапазоне, начинающемся с first, значение gen(), где gen — функциональный объект генератора, т.е. функция, не принимающая аргументов. Например, gen может быть указателем на функцию rand().

remove()

```
template<class ForwardIterator, class T>
ForwardIterator remove(ForwardIterator first, ForwardIterator last,
   const T& value);
```

Функция remove() удаляет все вхождения value в диапазоне [first, last) и возвращает для результирующего диапазона итератор для элемента, следующего за последним элементом. Эта функция является устойчивой, т.е. порядок не удаленных элементов остается неизмененным.

На заметку!

Поскольку различные функции remove() и unique() не являются функциями-членами и так как они не ограничены применением только к контейнерам библиотеки STL, они не могут переустанавливать размер контейнера. Вместо этого они возвращают итератор, который указывает на новую позицию, следующую за последней позицией. Обычно удаленные элементы просто сдвигаются в конец контейнера. Однако для контейнеров библиотеки STL можно использовать возвращаемый итератор и один из методов erase() для сброса end().

remove_if()

```
template<class ForwardIterator, class Predicate>
ForwardIterator remove_if(ForwardIterator first, ForwardIterator last,
   Predicate pred);
```

Функция remove_if() удаляет все вхождения значений val, для которых pred(val) дает true, из диапазона [first, last) и возвращает для результирующего диапазона итератор, который указывает на элемент, следующий за последним элементом. Функция является устойчивой, т.е. порядок не удаленных элементов остается неизмененным.

remove_copy()

```
template<class InputIterator, class OutputIterator, class T>
OutputIterator remove_copy(InputIterator first, InputIterator last,
                     OutputIterator result, const T& value);
```

Функция remove_copy() копирует значения из диапазона [first, last) в диапазон, начинающийся с result, пропуская при копировании экземпляры value. Функция возвращает для результирующего диапазона итератор, который указывает на элемент, следующий за последним элементом. Функция является устойчивой, т.е. порядок не удаленных элементов остается неизмененным.

remove_copy_if()

```
template<class InputIterator, class OutputIterator, class Predicate>
OutputIterator remove_copy_if(InputIterator first, InputIterator last,
                              OutputIterator result, Predicate pred);
```

Функция `remove_copy_if()` копирует значения из диапазона `[first, last)` в диапазон, начинающийся с `result`, пропуская при копировании экземпляры `val`, для которых `pred(val)` дает `true`. Функция возвращает для результирующего диапазона итератор, который указывает на элемент, следующий за последним элементом. Функция является устойчивой, т.е. порядок не удаленных элементов остается неизмененным.

unique()

```
template<class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class BinaryPredicate>
ForwardIterator unique(ForwardIterator first, ForwardIterator last,
                       BinaryPredicate pred);
```

Функция `unique()` сокращает каждую последовательность из двух или более эквивалентных элементов в диапазоне `[first, last)` до одного элемента, после чего для нового диапазона возвращает итератор, который указывает на элемент, следующий за последним элементом. В первой версии для сопоставления элементов используется операция `==` для типа значения. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката `pred`. Таким образом, элементы, на которые указывает `it1` и `it2`, совпадают, если результат `pred(*it1, *it2)` равен `true`.

unique_copy()

```
template<class InputIterator, class OutputIterator>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result);

template<class InputIterator, class OutputIterator, class BinaryPredicate>
OutputIterator unique_copy(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryPredicate pred);
```

Функция `unique_copy()` копирует элементы из диапазона `[first, last)` в диапазон, начинающийся с `result`, сокращая каждую последовательность из двух или более идентичных элементов до одного элемента. Для нового диапазона функция возвращает итератор, который указывает на элемент, следующий за последним элементом. В первой версии для сопоставления элементов используется операция `==`, определенная для типа значения. Во второй версии для сопоставления элементов применяется функциональный объект бинарного предиката `pred`. Таким образом, элементы, на которые указывает `it1` и `it2`, совпадают, если результат `pred(*it1, *it2)` равен `true`.

reverse()

```
template<class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

Функция `reverse()` изменяет порядок элементов на противоположный в диапазоне `[first, last)`, вызывая `swap(first, last - 1)` и т.д.

reverse_copy()

```
template<class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first,
   BidirectionalIterator last, OutputIterator result);
```

Функция `reverse_copy()` копирует элементы из диапазона [`first`, `last`) в диапазон, начинающийся с `result`, в обратном порядке. Указанные два диапазона не должны перекрываться.

rotate()

```
template<class ForwardIterator>
void rotate(ForwardIterator first, ForwardIterator middle,
          ForwardIterator last);
```

Функция `rotate()` циклически сдвигает элементы влево в диапазоне [`first`, `last`). Элемент в позиции `middle` перемещается в позицию `first`, элемент в позиции `middle + 1` — в позицию `first + 1` и т.д. Элементы, предшествующие `middle`, закольцовываются через конец контейнера, поэтому элемент в позиции `first` будет следовать за элементом `last - 1`.

rotate_copy()

```
template<class ForwardIterator, class OutputIterator>
OutputIterator rotate_copy(ForwardIterator first, ForwardIterator middle,
                     ForwardIterator last, OutputIterator result);
```

Функция `rotate_copy()` копирует элементы из диапазона [`first`, `last`) в диапазон, начинающийся с `result`, используя циклически сдвинутую последовательность, которая описана для функции `rotate()`.

random_shuffle()

```
template<class RandomAccessIterator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class RandomNumberGenerator>
void random_shuffle(RandomAccessIterator first, RandomAccessIterator last,
                RandomNumberGenerator& random);
```

Первая версия функции `random_shuffle()` тасует элементы в диапазоне [`first`, `last`). Распределение является нормальным, т.е. каждая возможная перетасовка исходного порядка будет равновероятной.

Вторая версия функции `random_shuffle()` тасует элементы в диапазоне [`first`, `last`). Распределение определяется функциональным объектом `random`. Для `n` элементов выражение `random(n)` должно возвращать значение из диапазона [`0`, `n`). В C++98 аргумент `random` был ссылкой lvalue; в C++11 он является ссылкой rvalue.

shuffle()

```
template<class RandomAccessIterator, class Uniform RandomNumberGenerator>
void shuffle(RandomAccessIterator first, RandomAccessIterator last,
          UniformRandomNumberGenerator&& rgen);
```

Функция `shuffle()` тасует элементы в диапазоне [`first`, `last`). Тип функционального объекта `rgen` должен соответствовать требованиям к генератору случайных чисел с нормальным распределением, как определено стандартом C++11. Генератор `rgen` задает распределение. При наличии `n` элементов выражение `rgen(n)` должно возвращать значение из диапазона [`0`, `n`).

is_partitioned() (C++11)

```
template<class InputIterator, class Predicate>
bool is_partitioned(InputIterator first,
                    InputIterator last, Predicate pred);
```

Функция `is_partitioned()` возвращает `true`, если диапазон пуст или секционирован с помощью `pred` — т.е. организован так, что все элементы, удовлетворяющие `pred`, предшествуют всем тем элементам, которые не удовлетворяют `pred`. В противном случае функция возвращает `false`.

partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator partition(BidirectionalIterator first,
                                BidirectionalIterator last, Predicate pred);
```

Функция `partition()` помещает каждый элемент, значение `val` которого является таким, что `pred(val)` дает `true`, перед всеми элементами, не удовлетворяющими этому условию. Функция возвращает итератор для позиции, следующей за последней позицией со значением, для которого результат функционального объекта предиката был равен `true`.

stable_partition()

```
template<class BidirectionalIterator, class Predicate>
BidirectionalIterator stable_partition(BidirectionalIterator first,
                                       BidirectionalIterator last,
                                       Predicate pred);
```

Функция `stable_partition()` помещает каждый элемент, значение `val` которого является таким, что `pred(val)` дает `true`, перед всеми элементами, которые не удовлетворяют этому условию. Эта функция сохраняет относительную упорядоченность внутри каждой из двух групп. Функция возвращает итератор для позиции, следующей за последней позицией со значением, для которого результат функционального объекта предиката был равен `true`.

partition_copy() (C++11)

```
template<class InputIterator, class OutputIterator1,
       class OutputIterator2, class Predicate>
pair<OutputIterator1, OutputIterator2> partition_copy(
    InputIterator first, InputIterator last,
    OutputIterator1 out_true, OutputIterator2 out_false,
    Predicate pred);
```

Функция `partition_copy()` копирует каждый элемент, значение `val` которого является таким, что `pred(val)` дает `true`, в диапазон, начинающийся с `out_true`, а остальные элементы — в диапазон, начинающийся с `out_false`. Она возвращает пару объектов, содержащих итератор для конца диапазона, который начинается с `out_true`, и итератор для конца диапазона, который начинается с `out_false`.

partition_point() (C++11)

```
template<class ForwardIterator, class Predicate>
ForewardIterator partition_point(ForwardIterator first,
                                 ForwardIterator last,
                                 Predicate pred);
```

Функция partition_point() требует, чтобы диапазон был секционирован с помощью pred. Она возвращает итератор для позиции, следующей за последней позицией со значением, для которого результат функционального объекта предиката был равен true.

Операции сортировки и связанные с ними операции

В табл. Ж.15 перечислены операции сортировки и связанные с ними операции. Аргументы в этой таблице не показаны, а перегруженные функции представлены только один раз. Каждая функция имеет версию, в которой используется операция < для упорядочения элементов, и версию, в которой для упорядочения элементов применяется функциональный объект сравнения. За таблицей следует более подробное их описание, включая прототипы. Таким образом, вы можете просмотреть таблицу, чтобы узнать, что выполняет определенная функция, а затем обратиться к детальной информации.

Таблица Ж.15. Операции сортировки и связанные с ними операции

Функция	Описание
sort()	Сортирует диапазон
stable_sort()	Сортирует диапазон, сохраняя относительную упорядоченность эквивалентных элементов
partial_sort()	Частично сортирует диапазон, при этом для первых n элементов производится полная сортировка
partial_sort_copy()	Копирует частично отсортированный диапазон в другой диапазон
is_sorted()	Возвращает true, если диапазон отсортирован (C++11)
is_sorted_until()	Возвращает последний итератор, для которого диапазон отсортирован (C++11)
nth_element()	Для заданного итератора на диапазон находит элемент, который мог быть здесь, если бы диапазон был отсортирован, и помещает сюда этот элемент
lower_bound()	Для заданного значения находит первую позицию в отсортированном диапазоне, перед которой может быть вставлено значение, сохраняя прежнюю упорядоченность
upper_bound()	Для заданного значения находит последнюю позицию в отсортированном диапазоне, перед которой может быть вставлено значение, сохраняя прежнюю упорядоченность
equal_range()	Для заданного значения находит самый большой поддиапазон отсортированного диапазона, такой, в котором перед любым его элементом может быть вставлено значение, не нарушая упорядоченности
binary_search()	Возвращает true, если отсортированный диапазон содержит значение, эквивалентное данному значению; в противном случае возвращает false
merge()	Объединяет два отсортированных диапазона в третий диапазон
inplace_merge()	Объединяет два последовательно отсортированных диапазона на месте

Окончание табл. Ж.15

Функция	Описание
includes()	Возвращает true, если каждый элемент из одной последовательности можно найти в другой последовательности
set_union()	Создает объединение двух последовательностей, т.е. последовательность, содержащую все элементы, представленные в каждой из последовательностей
set_intersection()	Создает пересечение последовательностей, т.е. последовательность, содержащую только те элементы, которые могут быть найдены в обеих последовательностях
set_difference()	Создает разность двух последовательностей, т.е. последовательность, содержащую только те элементы, которые могут быть найдены в первой последовательности, но не во второй
set_symmetric_difference()	Создает последовательность, содержащую элементы, которые могут быть найдены в одной из двух последовательностей, но не в каждой из них
make_heap	Преобразует диапазон в частично упорядоченное полное бинарное дерево
push_heap()	Добавляет диапазон в частично упорядоченное полное бинарное дерево
pop_heap()	Удаляет наибольший элемент в частично упорядоченном полном бинарном дереве
sort_heap()	Сортирует частично упорядоченное полное бинарное дерево
is_heap()	Возвращает true, если диапазон является частично упорядоченным полным бинарным деревом (C++11)
is_heap_until()	Возвращает последний итератор, для которого диапазон является частично упорядоченным полным бинарным деревом (C++11)
min()	Возвращает меньшее из двух значений или наименьший элемент в объекте initializer_list (C++11)
max()	Возвращает большее из двух значений или наибольший элемент в объекте initializer_list (C++11)
minmax()	Возвращает пару объектов, содержащую значения двух аргументов в порядке увеличения размера или наименьший и наибольший элементы в аргументе initializer_list (C++11)
min_element()	Находит первое вхождение наименьшего значения в диапазоне
max_element()	Находит первое вхождение наибольшего значения в диапазоне
minmax_element()	Возвращает пару объектов, содержащую итератор для первого вхождения наименьшего значения в диапазоне и итератор для последнего вхождения наибольшего значения в диапазоне (C++11)
lexicographic_compare()	Выполняет лексикографическое сравнение двух последовательностей, возвращая true, если первая последовательность лексикографически меньше второй; в противном случае возвращает false
next_permutation()	Генерирует следующую перестановку в последовательности
previous_permutation()	Генерирует предыдущую перестановку в последовательности

Функции, представленные в этом разделе, выясняют порядок двух элементов с помощью операции <, определенной для элементов, или посредством объекта сравнения, определяемого шаблонным типом Compare. Если comp является объектом Compare, то comp(a,b) является обобщенной формой a < b и возвращает true, если a предшествует b в схеме упорядочения. Если результат a < b равен false, и b < a также равен false, то a и b эквивалентны друг другу. Объект сравнения должен обеспечивать как минимум *строгое квазиупорядочение*. Это упорядочение определяется перечисленными ниже положениями.

- Выражение comp(a,a) должно иметь результат false, обобщая тот факт, что значение не может быть меньше самого себя (т.е. сравнение является строгим).
- Если результат comp(a,b) равен true и результат comp(b,c) также равен true, то результат comp(a,c) будет равен true (т.е. сравнение является транзитивным отношением).
- Если элемент a эквивалентен b, а элемент b эквивалентен c, то элемент a эквивалентен c (т.е. эквивалентность является транзитивным отношением).

Если операцию < выполнять над целыми числами, то под эквивалентностью будет подразумеваться равенство, однако для общих случаев это не всегда так. Например, можно определить структуру с несколькими членами, описывающими почтовые адреса, и затем определить объект сравнения comp, который будет упорядочивать структуры по почтовому индексу. Тогда любые два адреса, имеющие одинаковые почтовые индексы, могут быть эквивалентными, но не равными друг другу.

Теперь давайте рассмотрим более детально операции сортировки и связанные с ними операции. Для каждой функции показан прототип (прототипы) и приводится краткое пояснение. Этот раздел состоит из нескольких подразделов. Как и ранее, пары итераторов указывают на диапазоны, с выбранным именем шаблонного параметра, указывающего на тип итератора. Как обычно, диапазон определяется в виде [first, last), включая first и не включая last. Функции, передаваемые в виде аргументов, являются функциональными объектами, которые могут быть указателями или объектами с определенной операцией (). Как и в главе 16, предикат представляет собой булевскую функцию с одним аргументом, а бинарный предикат — булевскую функция с двумя аргументами. (Функции не обязательно должны иметь тип bool, поскольку они возвращают значение 0, соответствующее false, и ненулевое значение, соответствующее true.) Кроме этого, как упоминалось в главе 16, унарным функциональным объектом является функция, принимающая один аргумент, а бинарным функциональным объектом — функция, принимающая пару аргументов.

Сортировка

Для начала рассмотрим алгоритмы сортировки.

sort()

```
template<class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last,
        Compare comp);
```

Функция sort() выполняет сортировку по возрастанию элементов в диапазоне [first, last) с использованием для сравнения операции <, определенную для типа значения. Для определения порядка первая версия применяет <, а вторая — объект сравнения comp.

stable_sort()

```
template<class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last,
                 Compare comp);
```

Функция `stable_sort()` выполняет сортировку элементов в диапазонс `[first, last)`, сохраняя относительную упорядоченность эквивалентных элементов. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

partial_sort()

```
template<class RandomAccessIterator>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void partial_sort(RandomAccessIterator first, RandomAccessIterator middle,
                  RandomAccessIterator last, Compare comp);
```

Функция `partial_sort()` выполняет частичную сортировку элементов в диапазоне `[first, last)`. Первые `middle - first` элементов отсортированного диапазона помещаются в диапазон `[first, middle)`, а прочие элементы остаются несортированными. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

partial_sort_copy()

```
template<class InputIterator, class RandomAccessIterator>
RandomAccessIterator partial_sort_copy(InputIterator first, InputIterator last,
                                       RandomAccessIterator result_first,
                                       RandomAccessIterator result_last);
template<class InputIterator, class RandomAccessIterator, class Compare>
RandomAccessIterator
partial_sort_copy(InputIterator first, InputIterator last,
                  RandomAccessIterator result_first,
                  RandomAccessIterator result_last,
                  Compare comp);
```

Функция `partial_sort_copy()` копирует первые `n` элементов отсортированного диапазона `[first, last)` в диапазон `[result_first, result_first + n)`. Значение `n` является наименьшим из `last - first` и `result_last - result_first`. Функция возвращает `result_first + n`. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

is_sorted() (C++11)

```
template<class ForwardIterator>
bool is_sorted(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
bool is_sorted(ForwardIterator first, ForwardIterator last, Compare comp);
```

Функция `is_sorted()` возвращает `true`, если диапазон `[first, last)` отсортирован, и `false` — в противном случае. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

is_sorted_until() (C++11)

```
template<class ForwardIterator>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator is_sorted_until(ForwardIterator first, ForwardIterator last,
                                Compare comp);
```

Функция `is_sorted_until()` возвращает `last`, если диапазон `[first, last)` имеет менее двух элементов. Иначе она возвращает последний итератор `it`, для которого диапазон `[first, it)` отсортирован. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

nth_element()

```
template<class RandomAccessIterator>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last);

template<class RandomAccessIterator, class Compare>
void nth_element(RandomAccessIterator first, RandomAccessIterator nth,
                 RandomAccessIterator last, Compare comp);
```

Функция `nth_element()` находит элемент в диапазоне `[first, last)`, который мог быть расположен в позиции `nth` отсортированного диапазона, и помещает его в позицию `nth`. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

Бинарный поиск

Алгоритмы в группе бинарного поиска предполагают, что диапазон является отсортированным. Для них необходим только однонаправленный итератор, однако они являются самыми эффективными алгоритмами для случайных итераторов.

lower_bound()

```
template<class ForwardIterator, class T>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);
template<class ForwardIterator, class T, class Compare>
ForwardIterator lower_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

Функция `lower_bound()` находит первую позицию в отсортированном диапазоне `[first, last)`, перед которой значение `value` может быть вставлено без нарушения упорядочения. Функция возвращает итератор, указывающий на эту позицию. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

upper_bound()

```
template<class ForwardIterator, class T>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value);
template<class ForwardIterator, class T, class Compare>
ForwardIterator upper_bound(ForwardIterator first, ForwardIterator last,
                            const T& value, Compare comp);
```

Функция upper_bound() находит последнюю позицию в отсортированном диапазоне [first, last), перед которой значение value может быть вставлено без нарушения упорядочения. Для определения порядка первая версия использует <, а вторая — объект сравнения comp.

equal_range()

```
template<class ForwardIterator, class T>
pair<ForwardIterator, ForwardIterator> equal_range(
   ForwardIterator first, ForwardIterator last, const T& value);

template<class ForwardIterator, class T, class Compare>
pair<ForwardIterator, ForwardIterator> equal_range(
   ForwardIterator first, ForwardIterator last, const T& value, Compare comp);
```

Функция equal_range() находит наибольший поддиапазон [it1, it2) в отсортированном диапазоне [first, last), в котором значение value может быть вставлено перед любым итератором в этом диапазоне без нарушения упорядоченности. Функция возвращает пару pair, состоящую из it1 и it2. Для определения порядка первая версия использует <, а вторая — объект сравнения comp.

binary_search()

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                const T& value);
template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                const T& value, Compare comp);
```

Функция binary_search() возвращает true, если в отсортированном диапазоне [first, last) будет найдено значение, эквивалентное значению value; в противном случае функция возвращает false. Для определения порядка первая версия использует <, а вторая — объект сравнения comp.

> **На заметку!**
>
> Вспомните, что если для упорядочения используется операция <, то значения a и b будут эквивалентны друг другу, если сравнения a < b и b < a оба дают false. Для обычных чисел под эквивалентностью подразумевается равенство, однако, для структур, отсортированных на основе только одного члена, это не так. Таким образом, может существовать несколько позиций, в которые можно вставить новое значение с сохранением упорядоченности данных. Аналогично, если объект сравнения comp используется для формирования упорядоченности, то под эквивалентностью подразумевается, что сравнения comp(a,b) и comp(b,a) оба дают false. (Обобщенная форма утверждения, что a и b эквивалентны друг другу — если a не меньше b и b не меньше a.)

Слияние

Функции слияния работают с отсортированными диапазонами.

merge()

```
template<class InputIterator1, class InputIterator2,
       class OutputIterator>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result);
```

```
template<class InputIterator1, class InputIterator2,
       class OutputIterator, class Compare>
OutputIterator merge(InputIterator1 first1, InputIterator1 last1,
                 InputIterator2 first2, InputIterator2 last2,
                 OutputIterator result, Compare comp);
```

Функция `merge()` выполняет слияние элементов из отсортированного диапазона [`first1`, `last1`) и из отсортированного диапазона [`first2`, `last2`) с помещением результата в диапазон, начинающийся с `result`. Результирующий диапазон не должен перекрывать ни один из диапазонов, вовлеченных в слияние. Если в обоих диапазонах будут найдены эквивалентные элементы, то элементы из первого диапазона будут предшествовать элементам второго диапазона. Для результирующего слияния возвращаемым значением является итератор для элемента, следующего за последним элементом. Для определения порядка первая версия использует <, а вторая — объект сравнения `comp`.

`inplace_merge()`

```
template<class BidirectionalIterator>
void inplace_merge(BidirectionalIterator first,
   BidirectionalIterator middle, BidirectionalIterator last);

template<class BidirectionalIterator, class Compare>
void inplace_merge(BidirectionalIterator first, BidirectionalIterator middle,
   BidirectionalIterator last, Compare comp);
```

Функция `inplace_merge()` осуществляет слияние двух последовательно отсортированных диапазонов — [`first`, `middle`) и [`middle`, `last`) — в одну отсортированную последовательность, хранящуюся в диапазоне [`first`, `last`). Элементы из первого диапазона будут предшествовать эквивалентным элементам из второго диапазона. Для определения порядка первая версия использует <, а вторая — объект сравнения `comp`.

Работа с множествами

Операции с множествами могут выполняться над любыми отсортированными последовательностями, включая `set` и `multiset`. Для контейнеров, содержащих несколько экземпляров значения, например, `multiset`, определения обобщаются. Объединение двух мультимножеств содержит большее количество вхождений каждого элемента, а пересечение — меньшее количество вхождений каждого элемента. Предположим, например, что мультимножество `A` содержит семь строк `"apple"`, а мультимножество `B` — четыре таких строки. Объединение `A` и `B` будет содержать семь экземпляров строки `"apple"`, а пересечение — четыре упомянутых экземпляра.

`includes()`

```
template<class InputIterator1, class InputIterator2>
bool includes(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool includes(InputIterator1 first1, InputIterator1 last1,
           InputIterator2 first2, InputIterator2 last2, Compare comp);
```

Функция `includes()` возвращает `true`, если каждый элемент из диапазона [`first2`, `last2`) будет также найден в диапазоне [`first1`, `last1`); в противном случае функция возвращает `false`. Для определения порядка первая версия использует <, а вторая – объект сравнения `comp`.

set_union()

```
template<class InputIterator1, class InputIterator2,
       class OutputIterator>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result);
template<class InputIterator1, class InputIterator2,
       class OutputIterator, class Compare>
OutputIterator set_union(InputIterator1 first1, InputIterator1 last1,
                    InputIterator2 first2, InputIterator2 last2,
                    OutputIterator result, Compare comp);
```

Функция `set_union()` формирует множество, которое является объединением диапазонов [`first1`, `last1`) и [`first2`, `last2`), и копирует результат в позицию, на которую указывает `result`. Результирующий диапазон не должен перекрывать ни один из исходных диапазонов. Для результирующего диапазона функция возвращает итератор на элемент, следующий за последним элементом. Объединение представляет собой множество, состоящее из всех элементов, которые можно найти либо в одном из множеств, либо в обоих. Для определения порядка первая версия использует <, а вторая — объект сравнения `comp`.

set_intersection()

```
template<class InputIterator1, class InputIterator2,
       class OutputIterator>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          OutputIterator result);
template<class InputIterator1, class InputIterator2,
       class OutputIterator, class Compare>
OutputIterator set_intersection(InputIterator1 first1, InputIterator1 last1,
                          InputIterator2 first2, InputIterator2 last2,
                          OutputIterator result, Compare comp);
```

Функция `set_intersection()` формирует множество, которое представляет собой пересечение диапазонов [`first1`, `last1`) и [`first2`, `last2`), и копирует результат в позицию, на которую указывает `result`. Результирующий диапазон не должен перекрывать ни один из первоначальных диапазонов. Для сформированного диапазона функция возвращает итератор на элемент, следующий за последним элементом. Пересечение представляет собой множество, которое содержит элементы, общие для обоих множеств. Для определения порядка первая версия использует <, а вторая — объект сравнения `comp`.

set_difference()

```
template<class InputIterator1, class InputIterator2,
       class OutputIterator>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result);
template<class InputIterator1, class InputIterator2,
       class OutputIterator, class Compare>
OutputIterator set_difference(InputIterator1 first1, InputIterator1 last1,
                        InputIterator2 first2, InputIterator2 last2,
                        OutputIterator result, Compare comp);
```

Функция set_difference() формирует множество, представляющее собой разность между диапазонами [first1, last1) и [first2, last2), после чего копирует результат в позицию, на которую указывает result. Результирующий диапазон не должен перекрывать ни один из исходных диапазонов. Для сформированного диапазона функция возвращает итератор для элемента, следующего за последним элементом. Разность представляет собой множество, содержащее элементы, которые были найдены в первом множестве и не найдены во втором. Для определения порядка первая версия использует <, а вторая — объект сравнения comp.

set_symmetric_difference()

```
template<class InputIterator1, class InputIterator2,
       class OutputIterator>
OutputIterator set_symmetric_difference( InputIterator1 first1,
       InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result);
template<class InputIterator1, class InputIterator2,
       class OutputIterator, class Compare>
OutputIterator set_symmetric_difference(InputIterator1 first1,
       InputIterator1 last1,
       InputIterator2 first2, InputIterator2 last2,
       OutputIterator result, Compare comp);
```

Функция set_symmetric_difference() формирует последовательность, которая представляет собой симметричную разность между диапазонами [first1, last1) и [first2, last2), после чего копирует результат в позицию, на которую указывает result. Результирующий диапазон не должен перекрывать ни один из исходных диапазонов. Для сформированного диапазона функция возвращает итератор для элемента, следующего за последним элементом. Симметричная разность представляет собой множество, содержащее элементы, которые были найдены в первом множестве и не найдены во втором, и которые были найдены во втором множестве и не найдены в первом. Это то же самое, что и разность между объединением и пересечением. Для определения порядка первая версия использует <, а вторая — объект сравнения comp.

Работа с частично упорядоченными полными бинарными деревьями

Частично упорядоченное полное бинарное дерево (heap) — это общая форма представления данных, при которой первый элемент дерева является наибольшим элементом. Всякий раз, когда удаляется первый элемент или добавляется любой другой, может возникнуть необходимость в перегруппировке частично упорядоченного полного бинарного дерева с целью сохранения его свойства. Частично упорядоченное полное бинарное дерево создается таким образом, чтобы обеспечить эффективное выполнение этих двух операций.

make_heap()

```
template<class RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void make_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

Функция make_heap() создает частично упорядоченное полное бинарное дерево, определяемое диапазоном [first, last). Для определения порядка первая версия использует <, а вторая — объект сравнения comp.

push_heap()

```
template<class RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

Функция `push_heap()` предполагает, что диапазон `[first, last - 1)` является допустимым частично упорядоченным полным бинарным деревом и добавляет значение в позицию `last - 1` (т.е. пропускает конец дерева, которое предполагается как допустимое) частично упорядоченного полного бинарного дерева, в результате чего дерево `[first, last)` становится допустимым. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

pop_heap()

```
template<class RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last,
              Compare comp);
```

Функция `pop_heap()` предполагает, что диапазон `[first, last)` является допустимым частично упорядоченным полным бинарным деревом. Она осуществляет обмен значениями в позициях `last - 1` и `first`, в результате чего диапазон `[first, last - 1)` становится допустимым частично упорядоченным полным бинарным деревом. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

sort_heap()

```
template<class RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last,
               Compare comp);
```

Функция `sort_heap()` предполагает, что диапазон `[first, last)` является частично упорядоченным полным бинарным деревом, и осуществляет его сортировку. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

is_heap() (C++11)

```
template<class RandomAccessIterator>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last);
template<class RandomAccessIterator, class Compare>
bool is_heap(RandomAccessIterator first, RandomAccessIterator last,
             Compare comp);
```

Функция `is_heap()` возвращает `true`, если диапазон `[first, last)` является частично упорядоченным полным бинарным деревом, и `false` — в противном случае. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

is_heap_until() (C++11)

```
template<class RandomAccessIterator>
RandomAccessIterator is_heap_until(RandomAccessIterator first,
                                   RandomAccessIterator last);
```

```
template<class RandomAccessIterator, class Compare>
RandomAccessIterator is_heap_until(
   RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

Функция `is_heap_until()` возвращает `last`, если диапазон `[first, last)` имеет менее двух элементов. В противном случае она возвращает последний итератор `it`, для которого `[first, it)` диапазон является частично упорядоченным полным бинарным деревом. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

Поиск максимального и минимального значений

Функции нахождения минимума и максимума возвращают минимальное и максимальное значения пар значений и последовательностей значений.

`min()`

```
template<class T> const T& min(const T& a, const T& b);
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
```

Эти версии функции `min()` возвращают меньшее из двух значений. Если два значения эквивалентны, возвращается первое из них. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

```
template<class T> T min(initializer_list<T> t);
template<class T, class Compare>
T min(initializer_list<T> t), Compare comp);
```

Эти версии функции `min()` (C++11) возвращают наименьшее значение в списке инициализаторов `t`. Если два или более значений эквивалентны, возвращается копия первого вхождения этого значения. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

`max()`

```
template<class T> const T& max(const T& a, const T& b);
template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
```

Эти версии функции `max()` возвращают большее из двух значений. Если два значения эквивалентны, возвращается первое из них. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

```
template<class T> T max(initializer_list<T> t);
template<class T, class Compare>
T max(initializer_list<T> t), Compare comp);
```

Эти версии функции `max()` (C++11) возвращают наибольшее значение в списке инициализаторов `t`. Если два или более значений эквивалентны, возвращается копия первого вхождения этого значения. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

`minmax()` (C++11)

```
template<class T>
pair<const T&,const T&> minmax(const T& a, const T& b);
template<class T, class Compare>
pair<const T&,const T&> minmax(const T& a, const T& b, Compare comp);
```

Эти версии функции `minmax()` возвращают пару (`b`, `a`), если `b` меньше `a`, и пару (`a`, `b`) — в противном случае. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

```
template<class T> pair<T,T> minmax(initializer_list<T> t);
template<class T, class Compare>
pair<T,T> minmax(initializer_list<T> t), Compare comp);
```

Эти версии функции `minmax()` возвращают копии наименьшего и наибольшего элементов в списке инициализаторов `t`. Если несколько элементов эквивалентно наименьшему, возвращается первое вхождение. Если несколько элементов эквивалентно наибольшему, возвращается последнее вхождение. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

min_element()

```
template<class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last,
                    Compare comp);
```

Функция `min_element()` возвращает первый итератор `it` в диапазоне [`first`, `last`), такой, что ни один элемент из диапазона не будет меньше `*it`. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

max_element()

```
template<class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last,
Compare comp);
```

Функция `max_element()` возвращает первый итератор `it` в диапазоне [`first`, `last`), такой, что ни один элемент в диапазоне не будет больше `*it`. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

minmax_element() (C++11)

```
template<class ForwardIterator>
   pair<ForwardIterator,ForwardIterator>
      minmax_element(ForwardIterator first, ForwardIterator last);

template<class ForwardIterator, class Compare>
   pair<ForwardIterator,ForwardIterator>
      minmax_element(ForwardIterator first, ForwardIterator last,
                  Compare comp);
```

Функция `max_element()` возвращает пару объектов, содержащую первый итератор `it1` в диапазоне [`first`, `last`), такой, что ни один элемент из диапазона не будет меньше `*it1`, и первый итератор `it2` в диапазоне [`first`, `last`), такой, что ни один элемент из диапазона не будет больше `*it2`. Для определения порядка первая версия использует `<`, а вторая — объект сравнения `comp`.

lexicographical_compare()

```
template<class InputIterator1, class InputIterator2>
bool lexicographical_compare(
   InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2);

template<class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(
   InputIterator1 first1, InputIterator1 last1,
   InputIterator2 first2, InputIterator2 last2,
   Compare comp);
```

Функция `lexicographical_compare()` возвращает значение `true`, если последовательность элементов в диапазоне [`first1`, `last1`) лексикографически меньше, чем последовательность элементов в диапазоне [`first2`, `last2`); в противном случае функция возвращает `false`. При лексикографическом сравнении сравнивается первый элемент одной последовательности с первым элементом другой последовательности, т.е. сравниваются `*first1` с `*first2`. Если `*first1` меньше `*first2`, то функция возвращает `true`. Если `*first2` меньше `*first1`, функция возвращает `false`. Если они эквивалентны, сравнивается следующий элемент в каждой последовательности. Этот процесс продолжается до тех пор, пока не обнаружатся два неэквивалентных соответствующих элемента или не будет достигнут конец последовательности. Если две последовательности эквивалентны друг другу вплоть до завершения одной из последовательностей, то более короткая последовательность будет меньше. Если две последовательности эквивалентны и имеют одинаковую длину, функция возвращает `false`. Для определения порядка первая версия использует <, а вторая — объект сравнения `comp`. Лексикографическое сравнение является обобщенной формой алфавитного сравнения.

Работа с перестановками

Перестановкой последовательности называется изменение порядка элементов. Например, последовательность, состоящая из трех элементов, имеет шесть возможных вариантов упорядочения, поскольку в качестве первого элемента могут быть выбраны три элемента. Выбор определенного элемента для первой позиции оставляет возможность выбора двух элементов для второй позиции и одного элемента — для третьей. Например, шесть вариантов перестановки цифр 1, 2 и 3 выглядят следующим образом:

```
123 132 213 232 312 321
```

В общем случае последовательность, состоящая из n элементов, имеет

$$n \times (n-1) \times \ldots \times 1, \text{ или } n!,$$

возможных перестановок.

Функции перестановки подразумевают, что последовательность всех возможных перестановок может быть изменена в лексикографическом порядке, как было показано в предыдущем примере с шестью перестановками. В общем случае это означает, что существует определенная перестановка, предшествующая и следующая за каждой перестановкой. Например, 213 непосредственно предшествует 232, а 312 непосредственно следует за ней. Однако первая перестановка (123 в нашем примере) не имеет предшествующей, а последняя перестановка (321 в приведенном примере) не имеет последующей.

next_permutation()

```
template<class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
bool next_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

Функция `next_permutation()` преобразует последовательность в диапазоне [`first`, `last`) в следующую перестановку в лексикографическом порядке. Если следующая перестановка существует, то функция возвращает `true`. Если она не существует (т.е. диапазон содержит последнюю перестановку в лексикографическом порядке), то функция возвращает `false` и преобразует диапазон в первую перестановку в лексикографическом порядке. Для определения порядка первая версия использует <, а вторая — объект сравнения `comp`.

prev_permutation()

```
template<class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last);
template<class BidirectionalIterator, class Compare>
bool prev_permutation(BidirectionalIterator first,
                      BidirectionalIterator last, Compare comp);
```

Функция `previous_permutation()` преобразует последовательность в диапазоне [`first`, `last`) в предыдущую перестановку в лексикографическом порядке. Если предыдущая перестановка существует, то функция возвращает `true`. Если она не существует (т.е. диапазон содержит первую перестановку в лексикографическом порядке), функция возвращает `false` и преобразует диапазон в последнюю перестановку в лексикографическом порядке. Для определения порядка первая версия использует <, а вторая — объект сравнения `comp`.

Числовые операции

В табл. Ж.16 кратко описаны числовые операции из заголовочного файла `numeric`. Аргументы в этой таблице не показаны, а перегруженные функции представлены только один раз. Каждая функция имеет версию, в которой используется < для упорядочения элементов, и версию, в которой для упорядочения элементов применяется функциональный объект сравнения. За таблицей следует более подробное их описание, включая прототипы. Таким образом, вы можете просмотреть таблицу, чтобы узнать, что выполняет определенная функция, и затем обратиться к детальной информации о ней.

Таблица Ж.16. Числовые операции

Функция	Описание
`accumulate()`	Вычисляет накопленную сумму по значениям из диапазона
`inner_product()`	Вычисляет скалярное произведение двух диапазонов
`partial_sum()`	Копирует частичные суммы, вычисленные из одного диапазона, во второй диапазон
`adjacent_difference()`	Копирует смежные разности, вычисленные из элементов одного диапазона, в другой диапазон
`iota()`	Присваивает последовательные значения, такие как полученные с помощью операции ++, элементам в диапазоне (C++11)

Теперь давайте рассмотрим более подробно каждую из этих операций. Для каждой функции представлен прототип (прототипы) и дано краткое пояснение.

accumulate()

```
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

template <class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init,
             BinaryOperation binary_op);
```

Функция `accumulate()` присваивает `acc` значение `init`; затем она выполняет операцию `acc = acc + *i` (первая версия) или `acc = binary_op(acc, *i)` (вторая версия) для каждого итератора `i` в диапазоне `[first, last)` по порядку. Далее функция возвращает результирующее значение `acc`.

inner_product()

```
template <class InputIterator1, class InputIterator2, class T>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init);

template <class InputIterator1, class InputIterator2, class T,
class BinaryOperation1, class BinaryOperation2>
T inner_product(InputIterator1 first1, InputIterator1 last1,
                InputIterator2 first2, T init,
                BinaryOperation1 binary_op1, BinaryOperation2 binary_op2);
```

Функция `inner_product()` присваивает `acc` значение `init`; затем она выполняет операцию `acc = *i * *j` (первая версия) или `acc = binary_op(*i, *j)` (вторая версия) для каждого итератора `i` в диапазоне `[first1, last1)` по порядку и для каждого соответствующего итератора `j` в диапазоне `[first2, first2 + (last1 - first1))`. Другими словами, она вычисляет значение на основе первых элементов из каждой последовательности, затем на основе вторых элементов в каждой последовательности и т.д. до тех пор, пока не будет достигнут конец первой последовательности. (Следовательно, вторая последовательность должна иметь как минимум такую же длину, как у первой.) В завершение функция возвращает результирующее значение `acc`.

partial_sum()

```
template <class InputIterator, class OutputIterator>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator partial_sum(InputIterator first, InputIterator last,
                           OutputIterator result, BinaryOperation binary_op);
```

Функция `partial_sum()` присваивает `*first` результату `*result` или `*first + *(first + 1)` результату `*(result + 1)` (первая версия) либо присваивает `binary_op(first, *(first + 1))` результату `*(result + 1)` (вторая версия) и т.д. Другими словами, n-й элемент последовательности, начинающейся с `result`, содержит сумму (или эквивалент `binary_op`) первых `n` элементов последовательности, начинающейся с `first`. Функция возвращает итератор на элемент, следующий за последним элементом. Алгоритм допускает равенство `result` и `first`, т.е. результат можно копировать поверх исходной последовательности, если в этом возникает необходимость.

adjacent_difference()

```
template <class InputIterator, class OutputIterator>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result);

template <class InputIterator, class OutputIterator, class BinaryOperation>
OutputIterator adjacent_difference(InputIterator first, InputIterator last,
                        OutputIterator result, BinaryOperation binary_op);
```

Функция `adjacent_difference()` присваивает `*first` позиции `result` (`result = *first`). Последующим позициям в результирующем диапазоне присваиваются разности (или эквивалент `binary_op`) смежных позиций в исходном диапазоне. Другими словами, следующей позиции в искомом диапазоне `(result + 1)` присваивается `*(first + 1) - *first` (первая версия) или `binary_op(*(first + 1), *first)` (вторая версия) и т.д. Функция возвращает итератор на элемент, следующий за последним элементом. Алгоритм допускает равенство `result` и `first`, т.е. результат можно копировать поверх исходной последовательности, если в этом возникает необходимость.

iota() (C++11)

```
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T value);
```

Функция `iota()` присваивает значение `*first`, увеличивает `value`, как это делается с помощью `++value`, присваивает новое значение `value` следующему элементу в диапазоне и продолжает делать это вплоть до достижения элемента `last`.

3

Рекомендуемая литература и ресурсы в Интернете

Программированию на языке C++ посвящено множество хороших книг и ресурсов в Интернете. Ниже предложен список литературы, который следует рассматривать скорее как репрезентативный, а не полный. Помимо перечисленных, существует еще большое количество книг и сайтов. Но, тем не менее, этот список охватывает весьма широкий диапазон литературы.

Рекомендуемая литература

- Becker, Pete. *The C++ Standard Library Extensions.* Upper Saddle River, NJ: Addison-Wesley, 2007.

  В этой книге рассматривается библиотека C++ Library Technical Report (TR1). Она представляет собой дополнительную библиотеку для C++98, но большинство ее элементов были встроены в C++11. В книге, помимо прочего, описаны шаблоны неупорядоченных множеств, интеллектуальные указатели, библиотека регулярных выражений, библиотека генерации случайных чисел и кортежи.

- Гради Буч, Роберт А. Максимчук, Майкл У. Энгл, Бобби Дж. Янг, Джим Коналлен, Келли А. Хьюстон. *Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е издание*, пер. с англ., ИД "Вильямс", 2008 г.

  В этой книге изложены концепции объектно-ориентированного программирования (ООП), рассматриваются методы ООП и приводятся некоторые примеры приложений. Все примеры реализованы на языке C++.

- Cline, Marshall, Greg Lomow, and Mike Girou. *C++ FAQs, Second Edition.* Reading, MA: Addison-Wesley, 1998.

  Книга содержит большое количество ответов на часто задаваемые вопросы по C++.

- Josuttis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference.* Reading, MA: Addison-Wesley, 1999.

  В книге описывается библиотека Standard Template Library (STL) и функциональные средства библиотеки C++, например, поддержка комплексных чисел и потоков ввода-вывода.

- Karlsson, Björn. *Beyond the C++ Standard Library: An Introduction to Boost.* Upper Saddle River, NJ: Addison-Wesley, 2006.

  В этой книге рассматриваются многочисленные библиотеки Boost.

- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition.* Upper Saddle River, NJ:Addison-Wesley, 2005.

  Книга предназначена для программистов, уже знакомых с C++, и предлагает 55 правил и рекомендаций. Часть из них посвящена техническим вопросам. Например, в ней объясняется, когда необходимо определять конструкторы копирования и операции присваивания. Другая часть посвящена общим вопросам, например, отношениям *является* и *содержит*.

- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.* Reading, MA: Addison-Wesley, 2001.

  Руководство по выбору контейнеров и алгоритмов; рассматриваются и другие аспекты использования библиотеки STL.

- Meyers, Scott. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Reading, MA: Addison-Wesley, 1996.

  Эта книга продолжает традицию *Effective C++*, объясняя некоторые менее понятные аспекты языка программирования, и демонстрирует примеры реализации различных задач, например, проектирование интеллектуальных указателей. В ней собран опыт программистов C++ за последние несколько лет.

- Дэвид Р. Мюссер, Жилмер Дж. Дердж, Атул Сейни. *C++ и STL: справочное руководство, 2-е издание (серия C++ in Depth),* пер. с англ., ИД "Вильямс", 2010 г.

  Полноценная книга по STL, в которой рассматриваются и иллюстрируются ее возможности.

- Stroustrup, Bjarne. *The C++ Programming Language, Third Edition.* Reading, MA: Addison-Wesley, 1997.

  Страуструп — создатель C++. Если вы знакомы с C++, то вы должны были заметить, что эта книга наиболее часто упоминается в различных источниках. В ней не только описан язык программирования, но и предлагается множество примеров его использования и рассматривается методология ООП. По мере совершенствования языка выходили новые издания этой книги, а это издание включает описание элементов стандартной библиотеки, таких как STL и строк.

- Stroustrup, Bjarne. *The Design and Evolution of C++.* Reading, MA: Addison-Wesley, 1994.

  Если вас интересуют вопросы эволюции языка программирования C++, обратитесь к этой книге.

- Vandevoorde, David and Nocolai M. Jpsittos. *C++ Templates: The Complete Guide.* Reading, MA: Addison-Wesley, 2003.

  О шаблонах можно сказать очень много, о чем свидетельствует этот детальный справочник.

Ресурсы в Интернете

- Стандарт 2011 ISO/ANSI C++ Standard (ISO/IEC 14882:2011) доступен на ресурсах как Национального института стандартизации США (American National Standards Institute – ANSI), так и Международной организации по стандартизации (International Organization for Standardization – ISO).

  ANSI предлагает загружаемую электронную версию стандарта в формате PDF по цене $381. Ее можно заказать на сайте `http://webstore.ansi.org`.

  ISO предлагает загружаемую электронную версию стандарта в формате PDF по цене 352 швейцарских франка или версию на компакт-диске по той же цене на сайте `www.iso.org`.

  Цены могут измениться.

- Сайт C++ FAQ Lite посвящен часто задаваемым вопросам (на английском, китайском, французском, русском и португальском языках) и является упрощенной версией книги Клина (Cline) и других. На данный момент его можно найти по адресу `www.parashift.com/c++-faq-lite`.

- Модерируемая дискуссионная группа, посвященная вопросам по C++:

  `comp.lang.c++.moderated`

- Информацию по специфическим темам, связанным с C++, можно найти с помощью различных поисковых систем.

И
Переход к стандарту ANSI/ISO C++

Предположим, что у вас оказалась программа, написанная на языке C или более ранней версии C++, и вы хотите преобразовать ее код в соответствие со стандартом C++. В этом приложении вы найдете некоторые рекомендации по такому преобразованию. Часть из них посвящена переходу от C на C++, а другая часть связана с переходом от более ранних версий C++ к стандарту C++.

Используйте альтернативы для некоторых директив препроцессора

Препроцессор C/C++ предлагает множество директив. В общем случае в C++ принято использовать директивы, предназначенные для управления процессом компиляции, и не применять директивы в качестве замены кода. Например, директива `#include` является необходимым компонентом для управления файлами программы. Другие директивы, например, `#ifndef` и `#endif`, позволяют управлять процессом компиляции определенных блоков программы. Директива `#pragma` позволяет управлять параметрами определенного компилятора. Все эти директивы являются полезными, а в ряде случаев просто необходимыми инструментальными средствами. Однако следует соблюдать осторожность при использовании директивы `#define`.

Используйте `const` вместо `#define` для определения констант

Символические константы облегчают чтение и поддержку кода программы. Имена констант указывают на их назначение, и если вам нужно будет изменить значение, то для этого достаточно изменить его один раз в определении, а затем выполнить повторную компиляцию. В языке C для создания символических имен констант применяется препроцессор:

```
#define MAX_LENGTH 100
```

Препроцессор заменяет текст в исходном коде, подставляя 100 вместо MAX_LENGTH до начала компиляции.

В C++ для этой цели используется модификатор const в объявлении переменной:

```
const int MAX_LENGTH = 100;
```

В результате MAX_LENGTH будет интерпретироваться как константа только для чтения, имеющая тип int.

Использование модификатора const дает ряд преимуществ. Первым делом, в объявлении явным образом именуется тип. При работе с #define необходимо применять различные суффиксы для чисел, чтобы указать типы, отличные от char, int или double; например, необходимо использовать 100L, чтобы обозначить тип long, и 3.14F, чтобы обозначить тип float. Более важно то, что const можно без труда применять и для составных типов, как показано в следующем примере:

```
const int base_vals[5] = {1000, 2000, 3500, 6000, 10000};
const string ans[3] = {"yes", "no", "maybe"};
```

И, наконец, идентификаторы const подчиняются тем же правилам области видимости, что и переменные. Таким образом, можно создавать константы с глобальной областью видимости, областью видимости пространства имен и областью видимости блока. Если, скажем, определить константу в какой-то функции, можно будет не беспокоиться о конфликте определения с глобальной константой, используемой где-нибудь в программе. Например, рассмотрим следующий фрагмент:

```
#define n 5
const int dz = 12;
...
void fizzle()
{
    int n;
    int dz;
    ...
}
```

Препроцессор заменит

```
int n;
```

на

```
int 5;
```

и это приведет к ошибке компиляции. Однако переменная dz, определенная в fizzle(), будет локальной. Также при необходимости fizzle() может использовать операцию разрешения контекста (::) и обращаться к константе как ::dz.

Ключевое слово const в C++ позаимствовано из языка C, однако в версии C++ оно более полезно. Например, в C++ имеется внутреннее связывание внешних значений const, отличное от внешнего связывания по умолчанию, применяемого для переменных и const в языке C. Это означает, что каждый файл программы, использующий const, должен содержать это определение const. Хотя может потребоваться дополнительная работа, в действительности внутреннее связывание существенно облегчает жизнь. Посредством внутреннего связывания можно помещать определения const в заголовочный файл, который используется различными файлами в проекте. Для внешнего связывания такая схема вызовет ошибку компиляции, а для внутреннего связывания — нет. Кроме того, поскольку значение const необходимо определять в том файле, в котором оно используется (оно должно находиться в заголовочном файле,

который используется этим файлом), значения const можно применять в качестве аргументов размера массива:

```
const int MAX_LENGTH = 100;
...
double loads[MAX_LENGTH];
for (int i = 0; i < MAX_LENGTH; i++)
loads[i] = 50;
```

В C такой код работать не будет, поскольку объявление MAX_LENGTH может находиться в отдельном файле и быть недоступным при компиляции этого определенного файла. По правде говоря, следует упомянуть, что в языке C для создания констант с внутренним связыванием можно использовать модификатор static. Поскольку в C++ ключевое слово static применяется по умолчанию, об этом можно не вспоминать.

Между прочим, пересмотренный стандарт C (C99) позволяет использовать const в качестве спецификации размера массива, однако массив интерпретируется как новая форма массива, называемая *переменным массивом*, который не является частью стандарта C++.

Одна роль директивы #define по-прежнему является исключительно полезной — в качестве стандартной идиомы, используемой для управления компиляцией заголовочного файла:

```
// blooper.h
#ifndef _BLOOPER_H_
#define _BLOOPER_H_
// Здесь располагается код
#endif
```

Однако для обычных символических констант следует всегда использовать const вместо #define. Еще одна хорошая альтернатива, которую необходимо применять тогда, когда имеется совокупность связанных целочисленных констант, заключается в использовании enum:

```
enum {LEVEL1 = 1, LEVEL2 = 2, LEVEL3 = 4, LEVEL4 = 8};
```

Используйте inline вместо #define для определения коротких функций

Традиционный способ создания близкой к эквивалентной встроенной функции в C предусматривает использование макроопределения #define:

```
#define Cube(X) X*X*X
```

В результате препроцессор выполнит текстовую подстановку, при которой X заменяется соответствующим аргументом для Cube():

```
y = Cube(x);          // заменяет y = x*x*x;
y = Cube(x + z++);    // заменяет x + z++*x + z++*x + z++;
```

Поскольку препроцессор применяет текстовую подстановку вместо настоящей передачи аргументов, использование этих макроопределений может привести к неожиданным и некорректным результатам. Количество таких ошибок можно сократить, если в макроопределении применить множество круглых скобок для обеспечения корректного порядка выполнения операций:

```
#define Cube(X) ((X)*(X)*(X))
```

Однако даже такая форма не имеет дела со случаями применения таких значений, как z++.

Использование ключевого слова inline для обозначения встроенных функций в C++ является более надежным, поскольку при этом происходит настоящая передача аргументов. Более того, в качестве встроенных функций в C++ могут применяться обычные функции или методы класса:

```
class dormant
{
private:
   int period;
   ...
public:
   int Period() const { return period; }  // автоматически встроенная
   ...
};
```

Единственная положительная особенность макроопределения #define состоит в том, что в нем не указывается тип, поэтому его можно использовать для любого типа, для которого имеет смысл данная операция. В C++ можно создавать встроенные шаблоны, чтобы получить функции, не зависящие от типа, и при этом сохранить передачу аргументов.

Короче говоря, вместо макроопределений #define языка C следует применять встроенные функции C++.

Используйте прототипы функций

В действительности выбора здесь нет: в то время как прототипирование в C является необязательным, в C++ оно необходимо. Обратите внимание, что определение функции, такой как встроенная функция, перед первым использованием служит ее прототипом.

Применять const в прототипах функций и заголовках следует тогда, когда это необходимо. В частности, const должно использоваться с параметрами указателя и ссылочными параметрами, представляющими данные, которые не должны изменяться. Это не только позволит компилятору перехватывать ошибки, влекущие за собой изменение данных, но и сделает функцию более универсальной. Другими словами, функция с указателем или ссылкой const может обрабатывать как данные const, так и другие данные, а функция, которая не использует const с указателем или ссылкой, может обрабатывать только другие данные.

Используйте приведения типов

Одной из неприятных особенностей языка C, которые не нравились Страуструпу, является неупорядоченная операция приведения типа. Действительно, приведения типов часто применяются в программах, однако стандартное приведение типа является слишком неограниченным. Например, рассмотрим следующий фрагмент кода:

```
struct Doof
{
   double feeb;
   double steeb;
   char sgif[10];
};
Doof leam;
short * ps = (short *) & leam;      // старый синтаксис
int * pi = int * (&leam);           // новый синтаксис
```

В языке C ничто не помешает привести указатель одного типа к указателю совершенно несвязанного с ним типа.

В определенном отношении эта ситуация подобна истории с оператором `goto`. Проблема заключалась в том, что оператор `goto` оказался настолько гибким, что приводил к запутыванию кода. Решение состояло в том, чтобы для обработки обычных задач, в которых необходимо использовать `goto`, предоставить более ограниченные, структурированные версии этого оператора. В результате были разработаны такие языковые элементы, как циклы `for` и `while` и операторы `if else`. Стандартная версия C++ предлагает похожее решение проблемы неупорядоченного приведения типов, а именно – ограниченные приведения типов для обработки наиболее обычных ситуаций, в которых требуется выполнять такие приведения. Ниже перечислены операции приведения типов, которые были рассмотрены в главе 15:

```
dynamic_cast
static_cast
const_cast
reinterpret_cast
```

Таким образом, если приведение типа применяется к указателю, необходимо по возможности использовать одну из этих операций. В результате будет выполнено гарантированно корректное приведение типа.

Знакомьтесь с функциональными возможностями C++

Если вы применяли `malloc()` и `free()`, то вместо них следует использовать `new` и `delete`. Если для обработки ошибок вы применяли `setjmp()` и `longjmp()`, то вместо них следует обращаться к `try`, `throw` и `catch`. Старайтесь использовать тип `bool` для значений, представляющих `true` и `false`.

Используйте новую организацию заголовочных файлов

В стандарте C++ определены новые имена для заголовочных файлов, как упоминалось в главе 2. Если вы применяли заголовочные файлы в старом стиле, вам следует перейти к использованию имен в новом стиле. Это не просто косметическое изменение, наоборот – новые версии часто обладают новыми возможностями. Например, заголовочный файл `ostream` обеспечивает поддержку ввода и вывода для расширенных символов. Он также предлагает новые манипуляторы, такие как `boolalpha` и `fixed` (см. главу 17). По сравнению с функциями `setf()` или `iomanip` эти манипуляторы обеспечивают более простой интерфейс для настройки многих параметров форматирования. Если вы используете `setf()`, то при определении констант вместо `ios` следует указывать `ios_base`, т.е. необходимо применять `ios_base::fixed` вместо `ios::fixed`. Кроме того, новые заголовочные файлы включают пространства имен.

Используйте пространства имен

Пространства имен помогают организовать идентификаторы, используемые в программе, таким образом, чтобы избежать возникновения конфликтов имен. Поскольку стандартная библиотека, реализованная с помощью новой организации заголовочных

файлов, помещает имена в пространство имен std, то для работы с этими заголовочными файлами понадобится иметь дело с пространствами имен.

Для простоты в примерах из этой книги чаще всего применяется директива using, чтобы сделать доступными все имена из пространства имен std:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;      // директива using
```

Однако экспортирование всех имен из пространства, независимо от того, есть в этом необходимость или нет, противоречит целям пространств имен.

Предпочтительнее помещать директиву using внутрь функции, в результате чего имена будут доступны только в ее пределах.

Рекомендуется прибегнуть к еще лучшему варианту: чтобы сделать доступными те имена, которые необходимы программе, нужно использовать либо объявление using, либо операцию разрешения контекста (::). Например, следующий фрагмент кода делает доступными cin, cout и endl для остального кода в файле:

```
#include <iostream>
using std::cin;           // объявление using
using std::cout;
using std::endl;
```

С другой стороны, операция разрешения контекста делает имя доступным только в том выражении, в котором она применяется:

```
cout << std::fixed << x << endl; // использование операции разрешения контекста
```

Такой подход может показаться изнурительным, однако общие объявления using можно поместить в заголовочный файл:

```
// mynames -- заголовочный файл
using std::cin;           // объявление using
using std::cout;
using std::endl;
```

Продолжая этот процесс дальше, можно собрать объявления using в пространстве имен:

```
// mynames -- заголовочный файл
#include <iostream>
namespace io
{
   using std::cin;
   using std::cout;
   using std::endl;
}
namespace formats
{
   using std::fixed;
   using std::scientific;
   using std:boolalpha;
}
```

Затем в программу можно будет включить этот файл и использовать необходимые пространства имен:

```
#include "mynames"
using namespace io;
```

Используйте интеллектуальные указатели

Каждая операция `new` должна сопровождаться соответствующей операцией `delete`. Если работа функции, в которой используется операция `new`, прекращается из-за возникшего исключения, может возникнуть проблема. Как было сказано в главе 15, в случае использования объекта `autoptr` для отслеживания объекта, созданного `new`, операция `delete` активизируется автоматически. Появившиеся в C++11 объекты `unique_ptr` и `shared_ptr` предлагают даже более удобные альтернативы.

Используйте класс `string`

Традиционная строка в стиле C не привязана к какому-то реальному типу. Строку можно хранить в массиве символов и инициализировать массив символов строкой. Однако с помощью операции присваивания нельзя присвоить строку массиву символов; вместо этого необходимо применять `strcpy()` или `strncpy()`. С помощью условных операций невозможно произвести сравнение строк в стиле C; взамен придется использовать `strcmp()`. (Применение, например, операции > не приводит к синтаксической ошибке; просто будут сравниваться адреса строк, а не их содержимое.)

С другой стороны, класс `string` (см. главу 16 и приложение E) позволяет использовать объекты для представления строк. Для работы со строками определены операции присваивания, условные операции и операция сложения (для конкатенации). Более того, класс `string` поддерживает автоматическое управление памятью, поэтому обычно можно не беспокоиться о том, что будет введена строка, которая либо переполнит массив, либо будет усечена перед сохранением.

Класс `string` предоставляет множество удобных методов. Например, можно присоединять один объект `string` к другому, а также присоединять строку в стиле C или даже значение `char` к объекту `string`. Для функций, которые требуют аргумент строки в C, можно вызывать метод `c_str()` для возврата подходящего указателя на тип `char`.

Класс `string` предлагает не только надежный набор методов обработки задач, связанных со строками, вроде поиска подстрок, но также и средства, совместимые со стандартной библиотекой шаблонов (Standard Template Library — STL), поэтому к объектам `string` можно применять алгоритмы STL.

Используйте библиотеку STL

Библиотека STL (см. главу 16 и приложение Ж) предлагает готовые решения многих задач программирования, поэтому ее обязательно следует использовать. Например, вместо объявления массива объектов `double` или `string` можно создать объект `vector<double>` или `vector<string>`. Или, в случае C++11, если больше подходит массив фиксированного размера, необходимо использовать `array<double>` или `array<string>`. Преимущества такого подхода подобны преимуществам применения объектов `string` вместо строк в стиле C. Операция присваивания определена, поэтому ее можно использовать для присваивания одного объекта `vector` другому. Объект `vector` можно передавать по ссылке, а функция, получающая такой объект, может с помощью метода `size()` определить количество элементов в объекте `vector`. Встроенное управление памятью позволяет автоматически изменять размеры при использовании метода `pushback()` для добавления элементов в объект `vector`. И, естественно, библиотека предлагает множество полезных методов классов и обобщенные алгоритмы.

Если необходим список, двусторонняя очередь, стек, обычная очередь, множество или карта, вы должны обращаться к библиотеке STL, которая предоставляет полезные шаблоны контейнеров. Библиотека алгоритмов разработана так, что можно без труда копировать содержимое вектора в список или сравнивать содержимое множества с вектором. Библиотека STL спроектирована как набор инструментов, предлагающих базовые элементы, которые при необходимости можно включать в сборку.

Обширная библиотека алгоритмов разрабатывалась очень тщательно, поэтому вы можете достичь прекрасных результатов при относительно небольших усилиях в программировании. Концепция итератора, задействованная при реализации алгоритмов, означает, что алгоритмы можно применять не только к контейнерам STL, но и, например, к традиционным массивам.

К

Ответы на вопросы для самоконтроля

Ответы на вопросы для самоконтроля из главы 2

1. Они называются функциями.
2. Перед финальной компиляцией вместо этой директивы подставляется содержимое файла `ostream`.
3. Делает доступными для программы определения, созданные в пространстве имен `std`.
4. `cout << "Hello, world\n";`

   или

   `cout << "Hello, world" << endl;`
5. `int cheeses;`
6. `cheeses = 32;`
7. `cin >> cheeses;`
8. `cout << "We have " << cheeses << " varieties of cheese\n";`
9. Функция `froop()` вызывается с одним аргументом, который будет иметь тип `double`; функция будет возвращать значение типа `int`. Например, ее можно использовать следующим образом:

   ```
   int gval = froop(3.14159);
   ```

   Функция `rattle()` не имеет возвращаемого значения и ожидает аргумент типа `int`. Например, ее можно вызвать так:

   ```
   rattle(37);
   ```

   Функция `prune()` возвращает `int` и ожидает использования без аргумента. Например, ее можно вызвать следующим образом:

   ```
   int residue = prune();
   ```

10. Ключевое слово return не нужно использовать в функции, если она возвращает тип void. Однако его можно применять, если вы не предоставляете возвращаемого значения:

```
return;
```

11. Вероятной причиной является отсутствие доступа к заголовочному файлу iostream или к пространству имен std. Удостоверьтесь, что в начале файла присутствуют следующие операторы:

```
#include <iostream>
using namespace std;
```

Ответы на вопросы для самоконтроля из главы 3

1. Наличие более одного целочисленного типа позволяет выбрать такой тип, который наиболее точно подходит под конкретные требования. Например, можно было бы использовать short для экономии памяти или long для обеспечения нужного объема памяти либо увеличения скорости вычислений.

2.
```
short rbis = 80;                    // или short int rbis = 80;
unsigned int q = 42110;             // или unsigned q = 42110;
unsigned long ants = 3000000000; // или long long ants = 3000000000;
```

   Примечание: не следует считать, что тип int способен хранить значение 3000000000.

   Если ваша система поддерживает универсальную списковую инициализацию, можно использовать следующий код:

```
short rbis = {80};              // знак = не обязателен
unsigned int q {42110};         // можно было бы использовать = {42110}
long long ants {3000000000};
```

3. C++ не предоставляет автоматической защиты от превышения целочисленных пределов; чтобы узнать об ограничениях, можно воспользоваться заголовочным файлом climits.

4. Константа 33L имеет тип long, а константа 33 — тип int.

5. Эти два оператора не эквивалентны друг другу, хотя в некоторых системах результат их выполнения будет одинаковым. Более важно то, что первый оператор присваивает букву A переменной grade только в той системе, в которой используется код ASCII, а второй оператор работает для других кодировок. Кроме того, 65 является константой int, а 'A' — константой char.

6. Вот четыре способа:

```
char c = 88;
cout << c << endl;          // тип char выводит символ
cout.put(char(88));         // put() выводит char в виде символа
cout << char(88) << endl;  // новый стиль приведения значения к типу char
cout << (char)88 << endl; // старый стиль приведения значения к типу char
```

7. Ответ зависит от того, сколько байтов содержат эти типы. Если тип long занимает 4 байта, то потерь не будет. Это объясняется тем, что наибольшим значением типа long является примерно 2 миллиарда, что составляет 10 цифр.

Поскольку double предлагает как минимум 13 значащих цифр, округление не является необходимым. С другой стороны, тип long long может достигать 19 цифр, что превышает 13 значащих цифр, гарантированных для double.

8. а. 8 * 9 + 2 равно 72 + 2, т.е. 74
   б. 6 * 3 / 4 равно 18 / 4, т.е. 4
   в. 3 / 4 * 6 равно 0 * 6, т.е. 0
   г. 6.0 * 3 / 4 равно 18.0 / 4, т.е. 4.5
   д. 15 % 4 равно 3
9. Для решения первой задачи подойдет один из следующих вариантов:

```
int pos = (int) x1 + (int) x2;
int pos = int(x1) + int(x2);
```

   Чтобы сложить их как тип double и затем преобразовать, можно воспользоваться одним из следующих операторов:

```
int pos = (int) (x1 + x2);
int pos = int(x1 + x2);
```

10. а. int
    б. float
    в. char
    г. char32_t
    д. double

Ответы на вопросы для самоконтроля из главы 4

1. а. char actors[30];
   б. short betsie[100];
   в. float chuck[13];
   г. long double dipsea[64];
2. а. array<char,30> actors;
   б. array<short, 100> betsie;
   в. array<float, 13> chuck;
   г. array<long double, 64> dipsea;
3. int oddly[5] = {1, 3, 5, 7, 9};
4. int even = oddly[0] + oddly[4];
5. cout << ideas[1] << "\n"; // или << endl;
6. char lunch[13] = "cheeseburger"; // количество символов + 1

   или

```
char lunch[] = "cheeseburger";     // позволить компилятору
                                   // самостоятельно считать элементы
```

7. string lunch = "Waldorf Salad";

   или, если директива using отсутствует:

```
std::string lunch = "Waldorf Salad";
```

8.
```
struct fish {
   char kind[20];
   int weight;
   float length;
};
```

9.
```
fish petes =
   {
      "trout",
      12,
      26.25
   };
```

10.
```
enum Response {No, Yes, Maybe};
```

11.
```
double * pd = &ted;
cout << *pd << "\n";
```

12.
```
float * pf = treacle; // или = &treacle[0]
cout << pf[0] << " " << pf[9] << "\n";
// или использовать *pf и *(pf + 9)
```

13. В коде предполагается, что заголовочные файлы `iostream` и `vector` включены, а также присутствует директива `using`:

```
unsigned int size;
cout << "Enter a positive integer: ";
cin >> size;
int * dyn = new int [size];
vector<int> dv(size);
```

14. Да, этот код правильный. Выражение `"Home of the jolly bytes"` является строковой константой; следовательно, она оценивается как адрес начала строки. Объект `cout` интерпретирует адрес `char` как приглашение на вывод строки, однако приведение типа (`int *`) преобразует адрес к типу указателя на `int`, который впоследствии выводится в виде адреса. Короче говоря, этот оператор выводит адрес строки, предполагая, что тип `int` способен уместить адрес.

15.
```
struct fish
{
   char kind[20];
   int weight;
   float length;
};
fish * pole = new fish;
cout << "Enter kind of fish: ";
cin >> pole->kind;
```

16. В результате использования `cin >> address` программа будет пропускать пробельные символы, пока не обнаружит символ, отличный от пробельного. Затем она читает символы, пока снова не будет обнаружен пробельный. Таким образом, будет пропущен символ новой строки, следующий за числовым вводом, что избавляет от этой проблемы. С другой стороны, будет прочитано только одно слово, а не вся строка.

17.
```
#include <string>
#include <vector>
#include <array>
const int Str_num {10};   // or = 10
...
std::vector<std::string> vstr(Str_num);
std::array<std::string, Str_num> astr;
```

Ответы на вопросы для самоконтроля из главы 5

1. Цикл с проверкой на входе оценивает проверочное выражение до входа в тело цикла. Если условие изначально равно `false`, то тело цикла никогда не выполнится. Цикл с проверкой на выходе оценивает проверочное выражение после обработки тела цикла. Таким образом, тело цикла выполнятся один раз, даже если проверочное выражение изначально дает `false`. Циклы `for` и `while` являются циклами с проверкой на входе, а цикл `do while` — циклом с проверкой на выходе.

2. Будет напечатано следующее:

   ```
   01234
   ```

   Обратите внимание, что `cout << endl;` не является частью тела цикла (поскольку отсутствуют фигурные скобки).

3. Будет напечатано следующее:

   ```
   0369
   12
   ```

4. Будет напечатано следующее:

   ```
   6
   8
   ```

5. Будет напечатано следующее:

   ```
   k = 8
   ```

6. Проще всего воспользоваться операцией `*=`:

   ```
   for (int num = 1; num <= 64; num *= 2)
      cout << num << " ";
   ```

7. Операторы заключают в пару фигурных скобок для формирования одного составного оператора, или блока.

8. Да, первый оператор является правильным. Выражение `1,024` состоит из двух выражений, `1` и `024`, разделенных операцией запятой. Значение является значением выражения справа. Это `024`, которое является восьмеричным эквивалентом десятичного `20`, поэтому в объявлении переменной `x` присваивается значение `20`. Второй оператор также правилен. Однако в соответствии с приоритетом выполнения операций выражение будет оценено следующим образом:

   ```
   (y = 1), 024;
   ```

   То есть выражение слева устанавливает `y` в `1`, а значением всего выражения, которое не используется, является `024`, или `20`.

9. Форма `cin >> ch` пропускает пробелы, символы новой строки и табуляции. Две других формы читают эти символы.

Ответы на вопросы для самоконтроля из главы 6

1. Обе версии кода дают одинаковые ответы, однако версия 2 (с `if else`) более эффективна. Посмотрим, что произойдет, например, если `ch` является пробелом. В версии 1 после инкрементирования `spaces` выполняется проверка, является ли символ символом новой строки. Это непроизводительные затраты времени, поскольку код уже установил, что `ch` является пробелом и, следовательно, не может быть новой строкой. В версии 2 в аналогичной ситуации проверка на предмет новой строки пропускается.

2. `++ch` и `ch + 1` дают одно и то же числовое значение. Однако `++ch` имеет тип `char` и выводится в виде символа, в то время как `ch + 1`, поскольку добавляет `char` к типу `int`, является типом `int` и выводится в виде числа.

3. Поскольку в программе используется `ch = '$'` вместо `ch == '$'`, комбинированный ввод и вывод выглядит следующим образом:

```
Hi!
H$i$!$
$Send $10 or $20 now!
S$e$n$d$ $ct1 = 9, ct2 = 9
```

   Каждый символ преобразуется в `$` до вывода во второй раз. Также значение выражения `ch == $` является кодом символа `$`, следовательно, ненулевым, следовательно, `true`; таким образом, каждый раз происходит инкрементирование `ct2`.

4. а. `weight >= 115 && weight < 125`

   б. `ch == 'q' || ch == 'Q'`

   в. `x % 2 == 0 && x != 26`

   г. `x % 2 == 0 && !(x % 26 == 0)`

   д. `donation >= 1000 && donation <= 2000 || guest == 1`

   е. `(ch >= 'a' && ch <= 'z') ||(ch >= 'A' && ch <= 'Z')`

5. Не обязательно. Например, если `x` равно `10`, то `!x` равно `0`, а `!!x` равно `1`. Однако если `x` является переменной `bool`, то `!!x` является `x`.

6. `(x < 0)? -x : x`

   или

   `(x >= 0)? x : -x;`

7.
```
switch (ch)
{
   case 'A': a_grade++;
          break;
   case 'B': b_grade++;
          break;
   case 'C': c_grade++;
          break;
   case 'D': d_grade++;
          break;
   default:  f_grade++;
          break;
}
```

8. Если вы используете целочисленные метки, а пользователь вводит не целое число, такое как **q**, то программа зависнет, потому что целочисленный ввод не может быть обработан как символ. Однако если применяются целочисленные метки, а пользователь вводит целое число, например, **5**, то символьный ввод будет интерпретировать `5` как символ. Затем в части `default` оператора `switch` можно предложить ввести другой символ.

9. Вот одна из версий кода:

```
int line = 0;
char ch;
while (cin.get(ch) && ch != 'Q')
{
   if (ch == '\n')
   line++;
}
```

Ответы на вопросы для самоконтроля из главы 7

1. Эти три шага таковы: определение функции, предоставление прототипа и вызов функции.

2. а. `void igor(void);  // или void igor()`

   б. `float tofu(int n);  // или float tofu(int);`

   в. `double mpg(double miles, double gallons);`

   г. `long summation(long harray[], int size);`

   д. `double doctor(const char * str);`

   е. `void ofcourse(boss dude);`

   ж. `char * plot(map *pmap);`

3. ```
   void set_array(int arr[], int size, int value)
   {
      for (int i = 0; i < size; i++)
         arr[i] = value;
   }
   ```

4. ```
   void set_array(int * begin, int * end, int value)
   {
      for (int * pt = begin; pt != end; pt++)
         pt* = value;
   }
   ```

5. ```
   double biggest (const double foot[], int size)
   {
      double max;
      if (size < 1)
      {
         cout << "Invalid array size of " << size << endl;
         cout << "Returning a value of 0\n";
         return 0;
      }
   ```
   ```

```
    else // не обязательно, т.к. return завершает выполнение программы
    {
       max = foot[0];
       for (int i = 1; i < size; i++)
          if (foot[i] > max)
             max = foot[i];
       return max;
    }
}
```

6. Квалификатор const используется вместе с указателями для защиты от изменения исходных данных, на которые указывают указатели. Когда программа передает базовый тип, такой как int или double, она передает его по значению, поэтому функция работает с копией. Следовательно, исходные данные уже защищены.

7. Строка может быть сохранена в массиве char, представлена строковой константой в двойных кавычках, а также представлена с помощью указателя, указывающего на первый символ в строке.

8.
```
int replace(char * str, char c1, char c2)
{
     int count = 0;
     while (*str)     // пока не достигнут конец строки
     {
          if (*str == c1)
          {
               *str = c2;
               count++;
          }
          str++;      // переход к следующему символу
     }
     return count;
}
```

9. Поскольку C++ интерпретирует строку "pizza" как адрес ее первого элемента, применение операции * даст значение этого первого элемента, которым является символ p. Так как C++ интерпретирует строку "taco" как адрес ее первого элемента, то "taco"[2] будет рассматриваться как значение элемента, расположенного на две позиции дальше — т.е. символ c. Другими словами, строковая константа действует точно так же, как имя массива.

10. Для передачи структуры по значению нужно просто передать имя структуры glitz. Чтобы передать ее адрес, должна использоваться операция взятия адреса, т.е. &glitz. Передача по значению автоматически защищает исходные данные, однако отнимает время и расходует память. Передача по адресу экономит время и память, но не защищает исходные данные, если только для параметра функции не будет задан модификатор const. Кроме того, передача по значению означает возможность применения обычной нотации для членов структуры, а передача указателя — необходимость использования операции членства через указатель.

11.
```
int judge (int (*pf)(const char *));
```

12. а. Обратите внимание, что если ap — это структура applicant, то ap.credit_ratings — это имя массива, а ap.credit_ratings[i] — элемент массива.

```
void display(applicant ap)
{
   cout << ap.name << endl;
   for (int i = 0; i < 3; i++)
      cout << ap.credit_ratings[i] << endl;
}
```

б. Обратите внимание, что если ap — это структура applicant, то pa->credit_ratings — это имя массива, а pa->credit_ratings[i] — элемент массива.

```
void show(const applicant * pa)
{
   cout << pa->name << endl;
   for (int i = 0; i < 3; i++)
      cout << pa->credit_ratings[i] << endl;
}
```

13.
```
typedef void (*p_f1)(applicant *);
p_f1 p1 = f1;
typedef const char * (*p_f2)(const applicant *, const applicant *);
p_f2 p2 = f2;
p_f1 ap[5];
p_f2 (*pa)[10];
```

Ответы на вопросы для самоконтроля из главы 8

1. Хорошими кандидатами на то, чтобы быть встроенными функциями, являются короткие нерекурсивные функции, которые могут уместиться в одну строку кода.
2. а. `void song(const char * name, int times = 1);`

   б. Никаких. Информацию о значениях по умолчанию содержат только прототипы.

   в. Да, при условии сохранения значения по умолчанию для times:

```
void song(char * name = "O, My Papa", int times = 1);
```

3. Для вывода кавычки можно использовать либо строку "\"", либо символ '"'. В следующих функциях демонстрируются оба способа:

```
#include <iostream.h>
void iquote(int n)
{
   cout << "\"" << n << "\"";
}
void iquote(double x)
{
   cout << '"' << x << '"';
}
void iquote(const char * str)
{
   cout << "\"" << str << "\"";
}
```

4. а. Эта функция не должна изменять члены структуры, поэтому используется квалификатор const:

```
void show_box(const box & container)
{
   cout << "Made by " << container.maker << endl;
   cout << "Height = " << container.height << endl;
   cout << "Width = " << container.width << endl;
   cout << "Length = " << container.length << endl;
   cout << "Volume = " << container.volume << endl;
}
```

   б.
```
void set_volume(box & crate)
{
   crate.volume = crate.height * crate.width * crate.length;
}
```

5. Сначала измените прототипы следующим образом:

```
// Функция для изменения объекта array
void fill(std::array<double, Seasons> & pa);

// Функция, использующая объект array, но не изменяющая его
void show(const std::array<double, Seasons> & da);
```

Обратите внимание, что в show() должен использоваться квалификатор const для защиты объекта от изменения.

Далее, внутри main() измените вызов fill(), как показано ниже:

```
fill(expenses);
```

Вызов show() изменений не требует.

Затем приведите код функции fill() к следующему виду:

```
void fill(std::array<double, Seasons> & pa)  // изменена
{
   using namespace std;
   for (int i = 0; i < Seasons; i++)
   {
      cout << "Enter " << Snames[i] << " expenses: ";
      cin >> pa[i];     // изменена
   }
}
```

Обратите внимание, что вместо (*pa)[i] используется просто pa[i].

Наконец, show() требует только изменения в заголовке функции:

```
void show(std::array<double, Seasons> & da)
```

6. а. Это можно сделать за счет использования значения по умолчанию для второго аргумента:

```
double mass(double d, double v = 1.0);
```

Это также можно сделать с помощью перегрузки функции:

```
double mass(double d, double v);
double mass(double d);
```

б. Значение по умолчанию нельзя использовать для количества повторений, потому что значения по умолчанию должны предоставляться справа налево. В этом случае можно воспользоваться перегрузкой:

```
void repeat(int times, const char * str);
void repeat(const char * str);
```

в. Можно использовать перегрузку функции:

```
int average(int a, int b);
double average(double x, double y);
```

г. Это сделать не получится, поскольку оба варианта будут иметь одну и ту же сигнатуру.

7.
```
template<class T>
T max(T t1, T t2)  // или T max(const T & t1, const T & t2)
{
   return t1 > t2? t1 : t2;
}
```

8.
```
template<> box max(box b1, box b2)
{
   return b1.volume > b2.volume? b1 : b2;
}
```

9. v1 назначается тип float, v2 — тип float &, v3 — тип float &, v4 — тип int и v5 — тип double. Литерал 2.0 имеет тип double, поэтому произведение 2.0 * m также относится к double.

Ответы на вопросы для самоконтроля из главы 9

1. а. homer является автоматической переменной.

   б. secret должна быть определена как внешняя переменная в одном файле и объявлена с использованием extern — в другом.

   в. topsecret может быть определена как статическая переменная с внешним связыванием за счет помещения перед внешним определением ключевого слова static. Либо ее можно определить в неименованном пространстве имен.

   г. beencalled должна быть определена как локальная статическая переменная за счет помещения перед объявлением в функции ключевого слова static.

2. Объявление using делает доступным одиночное имя из пространства имен и имеет область видимости, соответствующую декларативной области, в которой встречается объявление using. Директива using делает доступными все имена из пространства имен. Применение директивы using равносильно объявлению имен в наименьшей декларативной области, которая содержит объявление using и само пространство имен.

3.
```
#include <iostream>
int main()
{
   double x;
   std::cout << "Enter value: ";
```

```
    while (! (std::cin >> x))
    {
       std::cout << "Bad input. Please enter a number: ";
       std::cin.clear();
       while (std::cin.get() != '\n')
          continue;
    }
    std::cout << "Value = " << x << std::endl;
    return 0;
}
```

4. Ниже показан переписанный код:

```
#include <iostream>
int main()
{
   using std::cin;
   using std::cout;
   using std::endl;
   double x;
   cout << "Enter value: ";
   while (! (cin >> x) )
   {
      cout << "Bad input. Please enter a number: ";
      cin.clear();
      while (cin.get() != '\n')
         continue;
   }
   cout << "Value = " << x << endl;
   return 0;
}
```

5. В каждом файле можно было бы иметь отдельные определения статической функции. Либо же в каждом файле можно было бы определить соответствующую функцию `average()` в неименованном пространстве имен.

6.
```
10
4
0
Other: 10, 1
another(): 10, -4
```

7.
```
1
4, 1, 2
2
2
4, 1, 2
2
```

Ответы на вопросы для самоконтроля из главы 10

1. Класс – это определение пользовательского типа. Объявление класса описывает способ хранения данных и методы (функции-члены класса), которые могут использоваться для доступа и манипулирования этими данными.

2. Класс представляет операции, которые можно выполнять над объектом класса посредством открытого интерфейса методов класса; это абстракция. Класс может использовать закрытую видимость (по умолчанию) для данных-членов, а это означает, что доступ к данным может быть осуществлен только через функции-члены; это сокрытие данных. Детали реализации, такие как представление данных и код методов, скрыты; это инкапсуляция.

3. Класс определяет тип, включая информацию о том, как он может быть использован. Объект представляет собой переменную или другой объект данных, вроде созданного с помощью new, который создается и используется в соответствии с определением класса. Между классом и объектом существует такое же отношение, как и между стандартным типом и переменной этого типа.

4. Если вы создаете несколько объектов заданного класса, то каждый объект будет обладать пространством памяти для хранения собственного набора данных. Но все объекты используют один набор функций-членов. (Обычно методы являются открытыми, а данные-члены – закрытыми, однако это вопрос политики, а не требований к классам.)

5. В этом примере для хранения символьных данных используются массивы char, однако вместо них можно применять объекты класса string.

```
// #include <cstring>

// определение класса
class BankAccount
{
private:
   char name[40];          // или std::string name;
   char acctnum[25];       // или std::string acctnum;
   double balance;
public:
   BankAccount(const char * client, const char * num, double bal = 0.0);
   // или BankAccount(const std::string & client,
   //                 const std::string & num, double bal = 0.0);
   void show(void) const;
   void deposit(double cash);
   void withdraw(double cash);
};
```

6. Конструктор класса вызывается при создании объекта этого класса или в случае явного обращения к конструктору. Деструктор класса вызывается после завершения работы с объектом.

7. Ниже показаны два возможных решения (обратите внимание на необходимость включения cstring или string.h для использования strncpy() или, в противном случае, включения string для работы с классом string):

```
BankAccount::BankAccount(const char * client, const char * num, double bal)
{
   strncpy(name, client, 39);
   name[39] = '\0';
   strncpy(acctnum, num, 24);
   acctnum[24] = '\0';
   balance = bal;
}
```

или

```
BankAccount::BankAccount(const std::string & client,
                         const std::string & num, double bal)
{
   name = client;
   acctnum = num;
   balance = bal;
}
```

Имейте в виду, что аргументы по умолчанию присутствуют в прототипе, но не в определении функции.

8. Конструктор по умолчанию либо не имеет аргументов, либо имеет значения по умолчанию для всех аргументов. Наличие конструктора по умолчанию позволяет объявлять объекты без их инициализации, даже если уже определен инициализирующий конструктор. Он также позволяет объявлять массивы.

9.
```
// stock30.h
#ifndef STOCK30_H_
#define STOCK30_H_
class Stock
{
private:
   std::string company;
   long shares;
   double share_val;
   double total_val;
   void set_tot() { total_val = shares * share_val; }
public:
   Stock();          // конструктор по умолчанию
   Stock(const std::string & co, long n, double pr);
   ~Stock() {}       // деструктор, который ничего не делает
   void buy(long num, double price);
   void sell(long num, double price);
   void update(double price);
   void show() const;
   const Stock & topval(const Stock & s) const;
   int numshares() const { return shares; }
   double shareval() const { return share_val; }
   double totalval() const { return total_val; }
   const string & co_name() const { return company; }
};
```

10. Указатель `this` доступен методам класса. Он указывает на объект, который был использован для вызова метода. Таким образом, `this` является адресом объекта, а `*this` представляет сам объект.

Ответы на вопросы для самоконтроля из главы 11

1. Ниже показан прототип для файла определения класса и определение функции для файла методов:

```
// прототип
Stonewt operator*(double mult);

// определение – позволяет конструктору выполнять свою работу
Stonewt Stonewt::operator*(double mult)
{
   return Stonewt(mult * pounds);
}
```

2. Функция-член является частью определения класса и вызывается конкретным объектом. Функция-член может обращаться к членам вызывающего объекта явным образом, не используя операцию членства. Дружественная функция не является частью класса, поэтому она вызывается подобно обычной функции. Она не может обращаться к членам класса явно, поэтому в ней должна применяться операция членства к объекту, переданному в качестве аргумента. Сравните, например, ответы на первый и четвертый вопросы.

3. Для доступа к закрытым членам она должна быть дружественной, но не должна быть таковой для обращения к открытым членам.

4. Ниже показан прототип для файла определения класса и определение функции для файла методов:

```
// прототип
friend Stonewt operator*(double mult, const Stonewt & s);

// определение – позволяет конструктору выполнять свою работу
Stonewt operator*(double mult, const Stonewt & s)
{
   return Stonewt(mult * s.pounds);
}
```

5. Не могут быть перегружены следующие пять операций:

```
sizeof
.
.*
::
?:
```

6. Эти операции должны быть определены с использованием функций-членов.

7. Ниже показан возможный вариант прототипа и определения:

```
// прототип и встроенное определение
operator double () {return mag;}
```

Однако обратите внимание, что лучше использовать метод `magval()`, чем определять эту функцию преобразования.

Ответы на вопросы для самоконтроля из главы 12

1. а. Синтаксис правильный, но этот конструктор оставляет неинициализированным указатель `str`. Конструктор должен либо установить указатель в `NULL`, либо использовать операцию `new []` для инициализации указателя.

   б. Этот конструктор не создает новую строку, а просто копирует адрес старой строки. Следует использовать `new []` и `strcpy()`.

   в. Этот конструктор копирует строку, не выделяя память для ее хранения. Чтобы выделить соответствующий объем памяти, необходимо использовать:

   ```
   new char[len + 1]
   ```

2. Во-первых, когда объект этого типа прекращает работу, данные, на которые указывает указатель-член объекта, остаются в памяти, занимая пространство и оставаясь недоступными, поскольку указатель был утрачен. Эту ситуацию можно исправить с помощью деструктора класса, который освободит память, выделенную операцией `new` в функциях конструкторов. Во-вторых, после того как деструктор освободит эту память, он может попытаться освободить ее еще раз, если в программе производилась инициализация одного объекта другим. Это объясняется тем, что при инициализации одного объекта другим по умолчанию копируется значение указателя, но не данные, на которые он указывает, в результате чего появляются два указателя на одни и те же данные. Решение состоит в том, чтобы определить конструктор копирования класса, благодаря которому при инициализации копировались бы данные, на которые указывает указатель. В-третьих, присваивание одного объекта другому может привести к аналогичной ситуации, когда два указателя будут указывать на одни и те же данные. Решение состоит в перегрузке операции присваивания, которая обеспечит копирование данных, а не указателей.

3. C++ автоматически предоставляет следующие функции-члены:

   - конструктор по умолчанию, если не определено ни одного конструктора;
   - конструктор копирования, если он не определен;
   - операция присваивания, если она не определена;
   - деструктор по умолчанию, если он не определен;
   - операция взятия адреса, если она не определена.

   Конструктор по умолчанию ничего не делает, однако позволяет объявлять массивы и неинициализированные объекты. Конструктор копирования и операция присваивания, предлагаемые по умолчанию, используют почленное присваивание. Деструктор по умолчанию ничего не делает. Неявная операция взятия адреса возвращает адрес вызывающего объекта (т.е. значение указателя `this`).

4. Член `personality` должен быть объявлен либо как массив символов, либо как указатель на `char`. Или же его можно сделать объектом `String` либо `string`. В объявлении эти методы не сделаны открытыми. В коде также присутствуют мелкие ошибки. Ниже показано одно из возможных решений, в котором изменения (отличные от удалений) выделены полужирным.

   ```
   #include <iostream>
   #include <cstring>
   ```

```
using namespace std;
class nifty
{
private:                    // необязательно
   char personality[40]; // предоставление размера массива
   int talents;
public:                     // обязательно
   // Методы
   nifty();
   nifty(const char * s);
   friend ostream & operator<<(ostream & os, const nifty & n);
}; // обратите внимание на завершающую точку с запятой

nifty::nifty()
{
   personality[0] = '\0';
   talents = 0;
}

nifty::nifty(const char * s)
{
   strcpy(personality, s);
   talents = 0;
}

ostream & operator<<(ostream & os, const nifty & n)
{
   os << n.personality << '\n';
   os << n.talent << '\n';
   return os;
}
```

А вот другое возможное решение:

```
#include <iostream>
#include <cstring>
using namespace std;
class nifty
{
private:                 // необязательно
   char * personality; // создание указателя
   int talents;
public:                  // обязательно
   // Методы
   nifty();
   nifty(const char * s);
   nifty(const nifty & n);
   ~nifty() { delete [] personality; }
   nifty & operator=(const nifty & n) const;
   friend ostream & operator<<(ostream & os, const nifty & n);
}; // обратите внимание на завершающую точку с запятой
```

```
nifty::nifty()
{
   personality = NULL;
   talents = 0;
}
nifty::nifty(const char * s)
{
   personality = new char [strlen(s) + 1];
   strcpy(personality, s);
   talents = 0;
}
ostream & operator<<(ostream & os, const nifty & n)
{
   os << n.personality << '\n';
   os << n.talent << '\n';
   return os;
}
```

5. a.
```
Golfer nancy;                  // конструктор по умолчанию
Golfer lulu("Little Lulu"); // Golfer(const char * name, int g)
Golfer roy("Roy Hobbs", 12); // Golfer(const char * name, int g)
Golfer * par = new Golfer;    // конструктор по умолчанию
Golfer next = lulu;           // Golfer(const Golfer &g)
Golfer hazard = "Weed Thwacker"; // Golfer(const char * name, int g)
*par = nancy;                 // операция присваивания по умолчанию
nancy = "Nancy Putter"; // Golfer(const char * name, int g), затем
                        // операция присваивания по умолчанию
```

   Обратите внимание, что некоторые компиляторы дополнительно вызывают операцию присваивания по умолчанию для операторов # 5 и #6.

   б. Класс должен определять операцию присваивания, которая копирует данные, а не адреса.

Ответы на вопросы для самоконтроля из главы 13

1. Открытые члены базового класса становятся открытыми членами производного класса. Защищенные члены базового класса становятся защищенными членами производного класса. Закрытые члены базового класса наследуются, но к ним невозможен доступ напрямую. В ответе на вопрос 2 рассматриваются исключения из этих общих правил.

2. Не наследуются методы конструктора, деструктор, операция присваивания и дружественные функции.

3. Если бы возвращаемым типом был `void`, по-прежнему можно было бы использовать одиночное присваивание, но не цепочку присваиваний:

```
baseDMA magazine("Pandering to Glitz", 1);
baseDMA gift1, gift2, gift3;
gift1 = magazine;          // нормально
gift 2 = gift3 = gift1;    // больше не допускается
```

Если метод возвращает вместо ссылки объект, то выполнение метода может несколько замедлиться, поскольку оператор возврата будет включать копирование объекта.

4. Конструкторы вызываются в порядке порождения, и первым вызывается наиболее вложенный в плане наследования конструктор. Деструкторы вызываются в обратном порядке.
5. Да, каждый класс требует собственных конструкторов. Если производный класс не добавляет новых членов, то конструктор может иметь пустое тело, но обязательно должен существовать.
6. Вызывается только метод производного класса. Он заменяет определение базового класса. Метод базового класса вызывается только тогда, когда производный класс не переопределяет метод или когда используется операция разрешения контекста. Однако в действительности необходимо объявлять виртуальной любую функцию, которая будет переопределена.
7. Производный класс должен определять операцию присваивания, если конструкторы производного класса используют операцию `new` или `new []` для инициализации указателей, которые являются членами данного класса. В общем случае производный класс должен определять операцию присваивания, если присваивание, предлагаемое по умолчанию, не подходит для членов производного класса.
8. Да, адрес объекта производного класса может быть присвоен указателю на базовый класс. Адрес объекта базового класса может быть присвоен указателю на производный класс (нисходящее преобразование) только путем явного приведения типа; использование такого указателя не всегда безопасно.
9. Да, объект производного класса может быть присвоен объекту базового класса. Однако любые члены данных, которые являются новыми по отношению к производному типу, базовому типу не передаются. Программа использует операцию присваивания базового класса. Присваивание в обратном направлении (объекта базового класса объекту производного класса) возможно, только если производный класс определяет операцию преобразования, которая является конструктором, имеющим ссылку на базовый тип в виде своего единственного аргумента, или если он определяет операцию присваивания с параметром базового класса.
10. Она может делать это, потому что C++ позволяет ссылке на базовый тип ссылаться на любой тип, который является производным от этого базового типа.
11. Передача объекта по значению активизирует конструктор копирования. Так как формальный аргумент является объектом базового класса, то вызывается конструктор копирования базового класса. Конструктор копирования получает в качестве своего аргумента ссылку на базовый класс, и эта ссылка может указывать на производный объект, передаваемый как аргумент. Совокупный результат состоит в том, что таким образом создается новый объект базового класса, члены которого соответствуют части базового класса в производном объекте.
12. Передача объекта по ссылке вместо передачи по значению позволяет использовать виртуальные функции. Кроме того, передача объекта по ссылке вместо передачи по значению может расходовать меньше памяти и занимать меньше времени, особенно для крупных объектов. Главное преимущество передачи по значению заключается в том, что при этом защищаются исходные данные, однако того же самого можно добиться и с помощью передачи ссылки как типа `const`.

13. Если `head()` является обычным мстодом, то `ph->head()` вызывает `Corporation::head()`. Если `head()` является виртуальной функцией, то `ph->head()` вызывает `PublicCorporation::head()`.

14. Во-первых, ситуация не соответствует модели *является*, поэтому открытое наследование не подходит. Во-вторых, определение `area()` в `House` скрывает версию `area()` в `Kitchen`, поскольку оба метода имеют разные сигнатуры.

Ответы на вопросы для самоконтроля из главы 14

1.

А	Б	
`class Bear`	`class PolarBear`	Открытый; белый медведь (polar bear) является разновидностью медведя (bear)
`class Kitchen`	`class Home`	Закрытый; в доме (home) имеется кухня (kitchen)
`class Person`	`class Programmer`	Открытый; программист (programmer) — это человек (person)
`class Person`	`class HorseAndJockey`	Закрытый; в связке лошадь и жокей (horse-jockey) присутствует человек (person)
`class Person`, `class Automobile`	`class Driver`	`Person` является открытым классом, поскольку водитель (driver) — это человек (person); `Automobile` является закрытым, поскольку у водителя есть автомобиль (automobile)

2.
```
Gloam::Gloam(int g, const char * s) : glip(g), fb(s) { }
Gloam::Gloam(int g, const Frabjous & fr) : glip(g), fb(fr) { }
// примечание: выше используется конструктор
//             копирования Frabjous по умолчанию
void Gloam::tell()
{
   fb.tell();
   cout << glip << endl;
}
```

3.
```
Gloam::Gloam(int g, const char * s)
    : glip(g), Frabjous(s) { }
Gloam::Gloam(int g, const Frabjous & fr)
    : glip(g), Frabjous(fr) { }
// примечание: выше используется конструктор
//             копирования Frabjous по умолчанию
void Gloam::tell()
{
   Frabjous::tell();
   cout << glip << endl;
}
```

4.
```
class Stack<Worker *>
{
private:
   enum {MAX = 10};        // константа, специфичная для класса
   Worker * items[MAX];    // хранит элементы стека
   int top;                // индекс вершины стека
public:
   Stack();
   Boolean isempty();
   Boolean isfull();
   Boolean push(const Worker * & item); // добавляет элемент в стек
   Boolean pop(Worker * & item);        // выталкивает элемент
                                        // с вершины стека
};
```

5.
```
ArrayTP<string> sa;
StackTP< ArrayTP<double> > stck_arr_db;
ArrayTP< StackTP<Worker *> > arr_stk_wpr;
```

   В листинге 14.18 генерируется четыре шаблона: `ArrayTP<int, 10>`, `ArrayTP< double, 10>`, `ArrayTP<int,5>` и `Array< ArrayTP<int,5>, 10>`.

6. Если две цепочки наследования класса имеют общего предка, то класс будет иметь две копии членов этого предка. Проблему можно решить, сделав класс предка виртуальным базовым классом по отношению к его непосредственным наследникам.

Ответы на вопросы для самоконтроля из главы 15

1. а. Объявление дружественного класса должно выглядеть следующим образом:

   ```
   friend class clasp;
   ```

   б. Для этого необходимо упреждающее объявление, чтобы компилятор мог интерпретировать `void snip(muff &)`:

   ```
   class muff;    // упреждающее объявление
   class cuff {
   public:
      void snip(muff &) { ... }
      ...
   };
   class muff {
      friend void cuff::snip(muff &);
      ...
   };
   ```

   в. Во-первых, объявление класса `cuff` должно предшествовать классу `muff`, чтобы компилятор мог воспринять `cuff::snip()`. Во-вторых, компилятору необходимо упреждающее объявление `muff`, чтобы он мог воспринять `snip(muff &)`:

   ```
   class muff;    // упреждающее объявление
   class cuff {
   public:
   ```

```
    void snip(muff &) { ... }
    ...
};
class muff {
    friend void cuff::snip(muff &);
    ...
};
```

2. Нет. Чтобы класс A имел друга, являющегося функцией-членом класса B, объявление B должно предшествовать объявлению A. Упреждающего объявления не будет достаточно, поскольку оно сообщит классу A, что B является классом, однако не будет показывать имена членов класса. Аналогично, если B имеет друга, который является функцией-членом A, то итоговое объявление A должно предшествовать объявлению B. Оба эти требования исключают друг друга.

3. Доступ к классу возможен только через его открытый интерфейс; это означает, что единственное, что вы можете сделать с объектом Sauce — это вызвать конструктор для его создания. Другие члены (soy и sugar) являются закрытыми по умолчанию.

4. Предположим, что функция f1() вызывает функцию f2(). Оператор возврата в f2() возобновляет выполнение программы со следующего оператора после вызова функции f2() в функции f1(). Оператор throw возвращает программу через существующую последовательность вызовов функций до блока try, который прямо или косвенно содержит вызов функции f2(). Он может находиться в f1() или в функции, вызывающей функцию f2(), и т.д. Отсюда выполнение передается следующему совпавшему блоку catch, а не первому оператору после вызова функции.

5. Блоки catch необходимо упорядочить от наиболее глубокого в цепочке наследования до наименее глубокого.

6. В примере # 1 условие if равно true, если pg указывает на объект Superb или на объект любого класса, унаследованного от Superb. В частности, оно также равно true, если pg указывает на объект Magnificient. В примере # 2 условие if равно true только для объекта Superb, а не для объектов, производных от Superb.

7. Операция dynamic_cast позволяет выполнять только восходящее приведение типов в иерархии класса, а операция static_cast — как восходящее, так и нисходящее приведение. Операция static_cast также позволяет выполнять преобразование перечислимых типов в целочисленные и наоборот, а также преобразование между различными числовыми типами.

Ответы на вопросы для самоконтроля из главы 16

1.
```
#include <string>
using namespace std;
class RQ1
{
private:
    string st; // объект string
public:
```

```
    RQ1() : st("") {}
    RQ1(const char * s) : st(s) {}
    ~RQ1() {};
    // дополнительный код
};
```

Явные конструктор копирования, деструктор и операция присваивания больше не нужны, поскольку объект `string` самостоятельно управляет памятью.

2. Один объект `string` можно присвоить другому. Объект `string` самостоятельно управляет памятью, поэтому обычно не нужно заботиться о превышении строкой объема памяти, которым обладает содержащий ее объект.

3.
```
#include <string>
#include <cctype>
using namespace std;
void ToUpper(string & str)
{
    for (int i = 0; i < str.size(); i++)
        str[i] = toupper(str[i]);
}
```

4.
```
auto_ptr<int> pia= new int[20];  // неправильно, необходимо
                                 // использовать new, а не new[]
auto_ptr<string>(new string);    // неправильно, отсутствует
                                 // имя указателя
int rigue = 7;
auto_ptr<int>(&rigue);           // неправильно, память не выделена
                                 // с помощью операции new
auto_ptr dbl (new double);       // неправильно, пропущено <double>
```

5. Принцип LIFO в стеке означает, что может понадобиться удалить множество клюшек для гольфа, прежде чем будет найдена необходимая.

6. Контейнер `set` будет хранить только одну копию каждого значения, поэтому, скажем, пять очков из 5 будут храниться как одно число 5.

7. Итераторы позволяют использовать объекты с интерфейсом, подобным указателям, для перемещения по данным, организованным не в виде массива (например, данные в двухсвязном списке).

8. Подход, реализованный в STL, позволяет функциям STL использоваться с обычными указателями на обычные массивы, а также с итераторами для контейнерных классов STL, подобным образом увеличивая их универсальность.

9. Можно присваивать один объект `vector` другому. Объект `vector` управляет собственной памятью, поэтому можно вставлять элементы в вектор, а он будет автоматически изменять свои размеры. С помощью метода `at()` можно инициировать автоматическую проверку границ.

10. Две функции `sort()` и функция `random_shuffle()` требуют итератора с произвольным доступом, тогда как объект `list` имеет только двунаправленный итератор. Для сортировки можно использовать функции-члены `sort()` шаблонного класса списка (см. приложение Ж) вместо функций общего назначения, однако нет функции-члена, эквивалентной `random_shuffle()`. Тем не менее, можно скопировать список в вектор, перетасовать вектор и скопировать результат обратно в список.

Ответы на вопросы для самоконтроля из главы 17

1. Файл iostream определяет классы, константы и манипуляторы, которые используются для управления вводом и выводом. Эти объекты управляют потоками и буферами, применяемыми при обработке ввода-вывода. Этот файл также создает стандартные объекты (cin, cout, cerr и clog и их эквиваленты для расширенных символов), которые используются для обработки стандартных потоков ввода и вывода, связанных с каждой программой.
2. При вводе с клавиатуры генерируется последовательность символов. При вводе **121** генерируются три символа, и каждый из них представляется однобайтным двоичным кодом. Если значение необходимо сохранить как int, то эти три символа должны быть преобразованы в одно двоичное представление значения 121.
3. По умолчанию стандартный вывод и стандартные ошибки отправляют вывод на стандартное устройство вывода, которым обычно является монитор. Однако если ваша операционная система перенаправит вывод в файл, то стандартный вывод будет связан с файлом, а не с экраном, но стандартные ошибки – с экраном.
4. Класс ostream определяет версию функции operator<<() для каждого базового типа C++. Компилятор интерпретирует выражение вроде

   ```
   cout << spot
   ```

   следующим образом:

   ```
   cout.operator<<(spot)
   ```

   Затем он может сопоставить этот вызов метода с прототипом функции, имеющей такой же тип аргумента.
5. Можно связать методы вывода, возвращающие тип ostream &. В результате метод будет вызываться посредством объекта для возврата этого объекта. Возвращенный объект впоследствии может активизировать следующий метод в последовательности.
6. ```
   // rq17-6.cpp
   #include <iostream>
   #include <iomanip>
   int main()
   {
      using namespace std;
      cout << "Enter an integer: ";
      int n;
      cin >> n;
      cout << setw(15) << "base ten" << setw(15)
           << "base sixteen" << setw(15) << "base eight" << "\n";
      cout.setf(ios::showbase);  // или cout << showbase;
      cout << setw(15) << n << hex << setw(15) << n
           << oct << setw(15) << n << "\n";
      return 0;
   }
   ```
   ```

7.
```
//rq17-7.cpp
#include <iostream>
#include <iomanip>
int main()
{
   using namespace std;
   char name[20];
   float hourly;
   float hours;
   cout << "Enter your name: ";
   cin.get(name, 20).get();
   cout << "Enter your hourly wages: ";
   cin >> hourly;
   cout << "Enter number of hours worked: ";
   cin >> hours;
   cout.setf(ios::showpoint);
   cout.setf(ios::fixed, ios::floatfield);
   cout.setf(ios::right, ios::adjustfield);
   // или cout << showpoint << fixed << right;
   cout << "First format:\n";
   cout << setw(30) << name << ": $" << setprecision(2)
        << setw(10) << hourly << ":" << setprecision(1)
        << setw(5) << hours << "\n";
   cout << "Second format:\n";
   cout.setf(ios::left, ios::adjustfield);
   cout << setw(30) << name << ": $" << setprecision(2)
        << setw(10) << hourly << ":" << setprecision(1)
        << setw(5) << hours << "\n";
   return 0;
```

8. Вывод программы выглядит следующим образом:

```
ct1 = 5; ct2 = 9
```

Первая часть программы игнорирует пробелы и символы новой строки, а вторая часть — нет. Обратите внимание, что вторая часть программы начинает чтение с символа новой строки, который следует за первым q, и воспринимает символ новой строки как часть строки.

9. Форма `ignore()` будет порождать сбойные ситуации, если строка ввода превысит 80 символов. В этом случае будут пропускаться только первые 80 символов.

Ответы на вопросы для самоконтроля из главы 18

1.
```
class Z200
{
private:
   int j;
   char ch;
   double z;
public:
   Z200(int jv, char chv, zv) : j(jv), ch(chv), z(zv) {}
   ...
};
```

```
double x {8.8};  // или = {8.8}
std::string s {"What a bracing effect!"};
int k{99};
Z200 zip{200,'Z',0.67});
std::vector<int> ai {3, 9, 4, 7, 1};
```

2. `r1(w)` является допустимым, и аргумент `rx` ссылается на `w`.

   `r1(w+1)` является допустимым, и аргумент `rx` ссылается на временный объект, инициализированный значением `w+1`.

   `r1(up(w))` является допустимым, и аргумент `rx` ссылается на временный объект, инициализированный возвращаемым значением `up(w)`.

   В общем случае, когда в качестве ссылочного lvalue-параметра `const` передается lvalue, параметр инициализируется этим значением lvalue. Если функции передается rvalue, ссылочный lvalue-параметр `const` ссылается на временную копию значения.

   `r2(w)` является допустимым, и аргумент `rx` ссылается на `w`.

   `r2(w+1)` является ошибочным, поскольку `w+1` — это rvalue.

   `r2(up(w))` является ошибочным, поскольку возвращаемое значение `up(w)` — это rvalue.

   В общем случае, когда в качестве ссылочного lvalue-параметра, отличного от `const`, передается lvalue, параметр инициализируется этим значением lvalue. Но ссылочный lvalue-параметр, отличный от `const`, не может принять аргумент функции rvalue.

   `r3(w)` является ошибочным, поскольку ссылка rvalue не может ссылаться на значение lvalue, такое как `w`.

   `r3(w+1)` является допустимым, и `rx` ссылается на временное значение выражения `w+1`.

   `r3(up(w))` является допустимым, и `rx` ссылается на временное возвращаемое значение `up(w)`.

3. а.
   ```
   double & rx
   const double & rx
   const double & rx
   ```

   Ссылка lvalue, отличная от `const`, соответствует lvalue-аргументу `w`. Другие два аргумента являются rvalue, и lvalue-ссылка `const` может ссылаться на их копии.

   б.
   ```
   double & rx
   double && rx
   double && rx
   ```

   Ссылка lvalue соответствует lvalue-аргументу `w`, и ссылки rvalue соответствуют двум аргументам rvalue.

   в.
   ```
   const double & rx
   double && rx
   double && rx
   ```

   lvalue-ссылка `const` соответствует lvalue-аргументу `w`, и ссылка rvalue соответствует двум значениям rvalue.

Короче говоря, параметр lvalue, отличный от `const`, соответствует аргументу lvalue, параметр rvalue, отличный от `const`, соответствует аргументу rvalue, а lvalue-параметр `const` может соответствовать либо аргументу lvalue, либо аргументу rvalue, но компилятор будет отдавать предпочтение первым двум вариантам, если они доступны.

4. Это конструктор по умолчанию, конструктор копирования, конструктор переноса, деструктор, операция присваивания с копированием и операция присваивания с перемещением. Они являются специальными потому, что компилятор может автоматически предоставить версии по умолчанию этих функций на основе контекста.

5. Конструктор переноса может использоваться, когда имеет смысл передача прав владения данными вместо их копирования, однако какие-либо механизмы передачи прав владения для стандартного массива не предусмотрены. Если бы в классе `Fizzle` применялись указатель и динамическое выделение памяти, то можно было бы передать права владения, присвоив адрес данных новому указателю.

6.
```
#include <iostream>
#include <algorithm>
template<typename T>
 void show2(double x, T fp) {std::cout << x << " -> " << fp(x) << '\n';}
int main()
{
   show2(18.0, [](double x){return 1.8*x + 32;});
   return 0;
}
```

7.
```
#include <iostream>
#include <array>
#include <algorithm>
const int Size = 5;
template<typename T>
   void sum(std::array<double,Size> a, T& fp);
int main()
{
   double total = 0.0;
   std::array<double, Size> temp_c = {32.1, 34.3, 37.8, 35.2, 34.7};
   sum(temp_c, [&total](double w){total += w;});
   std::cout << "total: " << total << '\n';
   std::cin.get();
   return 0;
}
template<typename T>
   void sum(std::array<double,Size> a, T& fp)
{
   for(auto pt = a.begin(); pt != a.end(); ++pt)
   {
      fp(*pt);
   }
}
```

Предметный указатель

К

Л

М

Н

О

П

Ф

Ц

Ч

Ш

Э

Я