
ДМК
издательство
C
для
программистов
С введением в C11
ПОЛ ДЕЙТЕЛ • ХАРВИ ДЕЙТЕЛ

Пол Дейтел, Харви Дейтел

C для программистов
с введением в C11

Paul Deitel, Harvey Deitel

C for Programmers
witn an Introduction in C11

PRENTICE
HALL

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

Пол Дейтел, Харви Дейтел

C для программистов
с введением в C11

ДМК ПРЕСС
издательство
Москва, 2014

УДК 004.438Си(075.8)
ББК 32.973-018.1я73-1
Д27

Дейтел П., Дейтел Х.
Д27 С для программистов с введением в С11 / пер. с анг. А. Киселева. – М.: ДМК Пресс, 2014. – 544 с.: ил.

ISBN 978-5-97060-073-3

В книге рассказывается о языке С и стандартной библиотеке С, следуя девизу компании Deitel: «обучение на действующих примерах». Понятия представляются в контексте тщательно протестированных программ, с выделением синтаксиса, демонстрацией вывода программ и подробного их обсуждения. Приводится примерно 5 000 строк кода на языке С и даются сотни полезных советов, которые помогут вам создавать надежные приложения.

Рассматривается создание собственных структур данных и стандартная библиотека, безопасное программирование на С; описываются особенности новой ревизии стандарта С11, в т. ч. многопоточность. Закончив чтение, вы будете иметь все знания, необходимые для создания приложений на языке С промышленного уровня.

Издание предназначено программистам, имеющим опыт работы на высокоуровневых языках.

УДК 004.438Си(075.8)
ББК 32.973-018.1я73-1

ISBN 978-0-13-346206-4 (анг.)
ISBN 978-5-97060-073-3 (рус.)

Памяти Дениса Ритчи (Dennis Ritchie),
создателя языка программирования C
и соавтора операционной системы UNIX.

Пол и Харви Дейтел

Содержание

Предисловие

Добро пожаловать в язык программирования C. Эта книга представляет самые современные вычислительные технологии для профессиональных программистов.

Основой этой книги является девиз компании Deitel: «Обучение на действующих примерах». Мы стараемся подавать описание понятий в контексте действующих программ, а не абстрактных фрагментов кода. Каждый пример кода сопровождается одним или несколькими примерами его выполнения. Прочитайте введение «Before You Begin» (прежде чем начать) по адресу `www.deitel.com/books/cfp/cfp_BYB.pdf`, где описывается, как подготовить компьютер к опробованию 130 примеров кода в этой книге и своих собственных программ на языке C. Все исходные тексты примеров можно загрузить по адресу: `www.deitel.com/books/cfp` или `www.pearsonhighered.com/deitel`. Используйте их в процессе чтения книги.

Эта книга представляет собой интересное и увлекательное введение в язык программирования C. Если у вас появятся вопросы, отправляйте их по адресу: `deitel@deitel.com` – мы постараемся ответить на них максимально быстро. Исправления и дополнения к книге можно найти по адресу: `www.deitel.com/books/cfp`, присоединяйтесь к нашим сообществам в социальных сетях: Facebook (`www.deitel.com/deitelfan`), Twitter (`@deitel`), Google+ (`gplus.to/deitel`) и LinkedIn (`bit.ly/deitelLinkedIn`), а также подпишитесь на рассылку новостей Deitel® Buzz Online (`www.deitel.com/newsletter/subscribe.html`).

Особенности

Ниже перечислены некоторые основные особенности книги «C для программистов с введением в C11».

- ***Информация подается с учетом положений нового стандарта языка C.*** Книга написана с учетом нового стандарта языка C, который часто называют *C11* или просто «стандарт C», потому что он вышел в 2011 году. Поддержка нового стандарта зависит исключительно от компилятора. Многие наши читатели используют либо компилятор GNU `gcc`, поддерживающий большинство ключевых особенностей нового стандарта, либо компилятор Microsoft Visual C++, который поддерживает лишь ограниченное подмножество особенностей, добавленных в язык C стандартами C99 и C11, в основном те, что также требуются стандартом языка C++. Чтобы не оставить за бортом ни одного из наших читателей, мы поместили обсуждение новых положений в необязательные разделы, которые без ущерба можно пропустить, и в приложение E «Дополнительные стандартные возмож-

ности в языке C». Мы также заменили все устаревшие особенности более новыми их версиями, чтобы обеспечить соответствие новому стандарту.

- ***Глава 1***. Представляет собой своеобразный тест-драйв, демонстрирующий порядок запуска консольных программ на языке C в Microsoft Windows, Linux и Mac OS X.
- ***Разделы о мерах безопасности при программировании на языке C***. Во многие главы добавлены замечания, касающиеся соблюдения мер безопасности при программировании на языке C. Эти же замечания были собраны нами на странице Secure C Programming Resource Center (`www.deitel.com/SecureC/`). Дополнительные подробности ищите в разделе «Замечания о мерах безопасности при программировании на языке C» ниже.
- ***Особое внимание уделяется проблемам производительности***. Язык C часто пользуется особым предпочтением у разработчиков программных продуктов, предъявляющих особые требования к производительности, таких как операционные системы, системы реального времени, встраиваемые системы и системы связи, поэтому проблемам производительности мы уделили особое внимание.
- ***Весь код проверен в Windows и Linux.*** Все примеры программ были протестированы с использованием компиляторов Visual C++® и GNU `gcc` в Windows и Linux соответственно.
- ***Сортировка под пристальным вниманием***. Сортировка – интереснейшая задача, потому что *один и тот же результат* можно получить самыми разными способами, которые могут существенно отличаться по таким параметрам, как производительность, потребление памяти и других системных ресурсов, – выбор алгоритма сортировки имеет важнейшее значение. Обсуждение проблем сортировки мы начнем в главе 6, а в приложении D уделим ей особое внимание. Мы познакомимся с разными алгоритмами и сравним их по потреблению памяти и производительности. С этой целью мы расскажем о нотации «большого O», которая описывает сложность алгоритмов, используемых для решения задач. В приложении D мы обсудим сортировку методом выбора (selection sort), сортировку методом вставок (insertion sort) и рекурсивную сортировку методом слияния (recursive merge sort).
- ***Приложения с описанием отладчиков***. Мы включили в книгу два приложения с описанием отладчиков Visual Studio® и GNU `gdb`.
- ***Порядок вычислений***. Мы обсудим некоторые тонкости, связанные с порядком вычислений, знание которых поможет вам избежать многих ошибок.

- ***Комментарии // в стиле C++***. В примерах мы используем новый и более краткий синтаксис комментариев в стиле языка `//`, отказавшись от устаревшего стиля `/*...*/`.
- ***Стандартная библиотека C***. Раздел 1.3 книги отсылает читателя к электронному справочнику `ru.cppreference.com/w/c`, где можно найти обширную документацию с описанием функций стандартной библиотеки C.

Замечания о мерах безопасности при программировании на языке C

Как показывает опыт, очень сложно писать промышленные системы, устойчивые к вирусам, червям и другим видам нападений. В настоящее время подобные нападения из Интернета могут происходить мгновенно и носить глобальный характер. Однако уязвимостей в программном обеспечении легко можно избежать. Если заниматься проблемами безопасности с самого начала разработки, это может значительно уменьшить трудозатраты на обеспечение безопасности и повысить устойчивость программного продукта к нападениям.

Для анализа нападений и противодействия им был создан координационный центр CERT® (`www.cert.org`). Группа CERT – Computer Emergency Response Team (группа реагирования на нарушения компьютерной безопасности) – издает и пропагандирует стандарты безопасного программирования, помогая программистам, использующим C и другие языки для создания промышленных систем, избежать применения приемов, открывающих системы для нападений. Группа CERT продолжает выпускать новые стандарты по мере выявления новых проблем безопасности.

Код наших примеров соответствует различным рекомендациям CERT на уровне, подходящем для демонстрации в книге. Если вам предстоит заниматься созданием промышленных систем на языке C, обязательно прочитайте книги Роберта Сикорда (Robert Seacord) «The CERT C Secure Coding Standard» (Addison-Wesley Professional, 2009) и «Secure Coding in C and C++» (Addison-Wesley Professional, 2013). Руководства CERT доступны в электронном виде по адресу `www.securecoding.cert.org`. Роберт Сикорд, научный редактор этой книги, также представил свои рекомендации в разделах «Безопасное программирование на C». Мистер Сикорд руководит отделом безопасного программирования в группе CERT института разработки программного обеспечения (Software Engineering Institute, SEI) университета Карнеги-Меллона (Carnegie Mellon University) и занимает пост адъюнкт-профессора в школе информатики университета Карнеги-Меллона.

В разделах «Безопасное программирование на C», в главах со 2 по 13, обсуждаются многие важные темы, включая тестирование арифметиче-

ских операций на переполнение; использование беззнаковых целочисленных типов; новые безопасные функции, определяемые стандартом Annex K; важность проверки информации, возвращаемой функциями из стандартной библиотеки; проверка диапазона; безопасное воспроизведение последовательности случайных чисел; проверка границ массивов; приемы предотвращения переполнения буфера; проверка ввода; исключение неопределенностей в поведении; предпочтение функций, возвращающих информацию о состоянии, перед аналогичными функциями, не возвращающими таких данных; гарантия сохранения в указателях допустимых адресов или значения `NULL`; предпочтение функций C перед макросами препроцессора и многие другие.

Подход к обучению

Книга «C для программистов с введением в C11» включает богатую коллекцию примеров. При создании примеров основное внимание уделялось надлежащей практике программирования и ясности программного кода.

Выделение синтаксиса. Для повышения читаемости примеров используется прием выделения синтаксиса, как это делает большинство сред разработки и редакторов для программистов. В этой книге используются следующие соглашения:

```
так оформляются ключевые слова, константы и литералы
так оформляется комментарии и весь остальной код
```

Использование шрифтов для выделения. Все ключевые термины в тексте книги выделяются **жирным** шрифтом. Элементы графического интерфейса приложений выделяются **рубленым шрифтом** (например, меню **File** (**Файл**)), а фрагменты программного кода – `моноширинным шрифтом` (например, `int x = 5;`).

Краткое описание целей. Каждая глава начинается с краткого описания целей главы.

Иллюстрации. В книге вы найдете множество диаграмм, таблиц, рисунков, блок-схем, исходного кода программ и примеров вывода этих программ.

Советы по программированию. В текст книги мы включили много советов и рекомендаций, чтобы обратить ваше внимание на особенно важные аспекты разработки программного обеспечения. Эти советы являются отражением 80-летнего (в сумме) опыта программирования и обучения.

Так будут оформляться советы и рекомендации, которые помогут вам создавать программы более простые, понятные и удобные для сопровождения.

Так будут оформляться описания распространенных ошибок программирования с целью помочь вам избежать их.

Так будут оформляться описания ошибок и способов их предотвращения. Во многих случаях будут описываться аспекты программирования на C, препятствующие появлению ошибок.

Так будут оформляться советы и рекомендации по повышению производительности приложений и уменьшению потребления памяти.

Так будут оформляться советы и рекомендации по повышению переносимости кода, который должен выполняться на разных платформах.

Так будут оформляться описания решений, влияющих на архитектуру программных систем, в первую очередь крупномасштабных.

Алфавитный указатель. В конце книги вы найдете обширный алфавитный указатель, который будет особенно полезен, когда вы будете использовать эту книгу в качестве справочника. В него включены основные термины с номерами страниц, где они встречаются.

Программное обеспечение, используемое в книге

Мы писали эту книгу, попутно используя свободно распространяемый компилятор GNU C (`gcc.gnu.org/install/binaries.html`), который предустановлен в большинстве систем Linux и может быть установлен в Mac OS X и Windows, а также бесплатную версию Microsoft Visual Studio Express 2012 for Windows Desktop (`www.microsoft.com/express`). Компилятор Visual C++ в Visual Studio способен компилировать программы на обоих языках, C и C++. Компания Apple включает компилятор LLVM в свой комплект инструментов разработки Xcode, который пользователи Mac OS X могут загрузить бесплатно из Mac App Store. Кроме того, в Интернете можно найти множество других бесплатных компиляторов C.

Основы языка C: видеоуроки, части I и II

В нашем комплекте видеоуроков «C Fundamentals: Parts I and II» (доступен с осени 2013 года) рассказывается обо всем, что необходимо знать, чтобы начать создавать надежные и мощные программы на языке C. Он включает видеоматериалы с лекциями общей продолжительностью 10+ часов, в ос-

нове которых лежат сведения из данной книги. За дополнительной информацией о видеоуроках из серии «Deitel LiveLessons» обращайтесь по адресу:

```
www.deitel.com/livelessons
```

или пишите на электронный адрес `deitel@deitel.com`. Доступ к видеоурокам из серии LiveLessons можно также получить, подписавшись на услугу Safari Books Online (`www.safaribooksonline.com`).

Благодарности

Мы хотим выразить благодарность Эбби Дейтел (Abbey Deitel) и Барбаре Дейтел (Barbara Deitel), что посвятили массу времени этому проекту. Нам повезло работать с командой опытных профессионалов и знатоков издательского дела из издательства Prentice Hall/Pearson. Мы высоко ценим невероятные усилия и помощь нашего друга и профессионала, сотрудничающего с нами вот уже 17 лет, Марка Л. Тауба (Mark L. Taub), шеф-редактора из Pearson Technology Group. Кэрол Снайдер (Carole Snyder) превосходно справилась с организацией процесса редактирования. Чути Прасертсих (Chuti Prasertsith) создал оригинальную обложку для книги. Джон Фуллер (John Fuller) давно и плодотворно занимается организацией издания всех наших книг из серии «Deitel®: Библиотечка разработчика».

Рецензенты

Мы весьма благодарны нашим рецензентам, которые, несмотря на весьма ограниченные сроки, тщательно проверили текст книги и исходный код примеров, и внесли бесчисленное множество ценных советов по их улучшению: доктор Джон Ф. Дойл (Dr. John F. Doyle, юго-восточное отделение университета штата Индиана), Хеманс Х.М. (Hemanth H.M., программист из SonicWALL), Витаутас Леонавичус (Vytautus Leonavicius, Microsoft), Роберт Сикорд (Robert Seacord, руководитель отдела безопасного программирования в SEI/CERT, автор «The CERT C Secure Coding Standard» и технический эксперт в рабочей группе по международной стандартизации языка программирования C) и Хосе Антонио Гонсалес Секо (José Antonio González Seco, парламент Андалусии, Испания).

Хорошо, что вы есть! C11 – мощный язык программирования, который поможет вам быстро и эффективно писать высокопроизводительные программы. C11 прекрасно подходит для разработки систем уровня предприятия и позволяет организациям создавать свои мощные информационные системы. Нам интересно будет знать ваше мнение об этой книге. Свои предложения, замечания и исправления вы можете направлять по адресу:

```
deitel@deitel.com
```

Мы постараемся ответить максимально быстро, а свои ответы и разъяснения мы разместим по адресу:

```
www.deitel.com/books/cfp
```

Мы надеемся, вам понравится книга «C для программистов с введением в C11» так же, как нам понравилось писать ее!

Пол и Харви Дейтел

Об авторах

Пол Дейтел (Paul Deitel), исполнительный и технический директор компании Deitel & Associates, Inc., окончил Массачусетский технологический институт, где изучал информационные технологии. Работая в Deitel & Associates, Inc., прочитал сотни курсов по программированию на промышленных предприятиях, в правительственных и военных организациях, включая Cisco, IBM, Siemens, Sun Microsystems, Dell, Fidelity, NASA (в космическом центре имени Кеннеди), национальную лабораторию исследования ураганов (США), полигон «White Sands Missile Range», RogueWave Software, Boeing, SunGard Higher Education, Nortel Networks, Puma, iRobot, Invensys и многие другие. Он и его соавтор, доктор Харви М. Дейтел (Harvey M. Deitel), являются самыми востребованными авторами книг и видеоуроков, посвященных различным языкам программирования.

Доктор Харви Дейтел (Harvey Deitel), президент и директор по стратегическим вопросам компании Deitel&Associates, Inc., обладает более чем 50-летним опытом работы в области вычислительной техники. Доктор Дейтел имеет степень бакалавра и магистра электротехники (применение вычислительной техники в обучении) Массачусетского технологического института, а также степень доктора математических наук, которую он защитил в Бостонском университете. Имеет огромный опыт преподавания, в том числе и на посту руководителя кафедры информатики в Бостонском колледже, который он занимал до того, как вместе со своим сыном Полом Дейтелом основал компанию Deitel & Associates, Inc., в 1991 году. Доктор Дейтел организовал сотни семинаров по программированию в крупных корпорациях, академических институтах, правительственных и военных организациях. Труды доктора Дейтела имеют международное признание и переведены на множество языков, в том числе китайский, корейский, японский, немецкий, русский, испанский, французский, польский, итальянский, португальский, греческий, урду и турецкий.

Корпоративные тренинги от Deitel & Associates, Inc.

Компания Deitel & Associates, Inc., основанная Полом и Харви Дейтел, известна во всем мире своими корпоративными тренингами и услугами по организации разработки программного обеспечения. Она специализиру-

ется на преподавании языков программирования, объектных технологий, особенностей разработки приложений для Android и iOS и веб-технологий. Компания предлагает проведение курсов у клиентов, независимо от страны, по изучению основных языков программирования и платформ, включая C, C++, Visual C++®, Java™, Visual C#®, Visual Basic®, XML®, Python®, объектные технологии, веб-программирование, разработку приложений для Android™, разработку приложений на Objective-C для iOS® и еще огромное количество курсов по разработке программного обеспечения. В число клиентов Deitel & Associates, Inc., входят крупнейшие мировые компании, а также правительственные и военные организации, академические институты.

За 37-летний промежуток сотрудничества с издательством Prentice Hall/Pearson, Deitel & Associates, Inc., опубликовала множество книг по профессиональному программированию, учебников и видеокурсов. Обратиться со своими предложениями к авторам книги и в компанию Deitel & Associates, Inc., можно по адресу:

```
deitel@deitel.com
```

Получить более подробную информацию о серии корпоративных тренингов Deitel Dive-Into® можно на сайте:

```
www.deitel.com/training
```

Заявки на проведение тренингов с преподавателем в своей организации вы можете направлять по электронному адресу: deitel@deitel.com.

Эта книга также доступна в электронном виде подписчикам услуги Safari Books Online по адресу:

```
www.safaribooksonline.com
```

Желающие в индивидуальном порядке приобрести книги и подписаться на видеокурсы компании Deitel могут сделать это на сайте. Оптовые заказы компаний, правительственных и военных организаций, а также академических институтов должны направляться непосредственно в издательство Pearson. За дополнительной информацией обращайтесь по адресу:

```
www.informit.com/store/sales.aspx
```

1

Введение

В этой главе вы познакомитесь:

- с историей развития языка C;
- с назначением стандартной библиотеки языка C;
- с элементами типичной среды разработки программ на языке C;
- с небольшим игровым приложением, выполняющимся в Windows, Linux и Mac OS X;
- с коммерческими и свободно распространяемыми операционными системами для компьютеров и смартфонов, для которых можно писать приложения на языке C.

Краткое содержание

1.1 Введение

Добро пожаловать в язык C – выразительный и мощный язык программирования, который с успехом можно использовать для создания крупных программных систем. Книга «C для программистов» в этом отношении является отличным учебником. Она подчеркивает эффективность структурного подхода к разработке программного обеспечения и включает 130 примеров законченных программ с результатами их выполнения. Мы называем это «обучением на действующих примерах». Исходный код всех программ, описываемых в книге, можно получить по адресу: `www.deitel.com/books/cfp/`.

1.2 Язык программирования C

Свою родословную язык C ведет от двух языков: BCPL и B. Язык BCPL был создан в 1967 году Мартином Ричардсом (Martin Richards) как язык для создания операционных систем и компиляторов. Кен Томпсон (Ken Thompson) заимствовал из него многие особенности при создании своего языка B, и в 1970 году он использовал B для создания первых версий UNIX в Bell Laboratories.

Дальнейшим продолжением языка B стал язык C, разработанный Денисом Ритчи (Dennis Ritchie), сотрудником Bell Laboratories, в 1972 году. Первоначально язык C был широко известен как язык разработки операционной системы UNIX. Многие современные операционные системы также написаны на C и/или C++. Язык C практически независим от аппаратной архитектуры – при надлежащем подходе к проектированию он позволяет писать программы, способные выполняться на самых разных аппаратных платформах.

Высокая производительность

Язык C широко используется для создания систем, где требуется высокая производительность, таких как операционные системы, встраиваемые системы, системы реального времени и системы связи (табл. 1.1).

В конце 70-х годов прошлого столетия язык C развился в то, что теперь называют «традиционный C». Книга Кернигана и Ритчи «The C Programming Language»[1], вышедшая в 1978 году, привлекла широкое внимание к языку. Она стала самой продаваемой книгой по информатике из тех, что когда-либо выпускались.

Таблица 1.1 | Некоторые приложения на языке C, с высокими требованиями к производительности

Приложение	Описание
Операционные системы	Переносимость и производительность языка C являются одними из самых востребованных его характеристик при реализации операционных систем, таких как Linux и Microsoft Windows, а также Google Android. Операционная система OS X, выпускаемая компанией Apple, написана на языке Objective-C, который корнями уходит в язык C. Некоторые основные операционные системы для компьютеров и мобильных устройств мы обсудим в разделе 1.7
Встраиваемые системы	Кроме компьютеров общего назначения, каждый год выпускается огромное количество микропроцессоров для встраиваемых устройств. К числу встраиваемых систем относятся системы для навигационных устройств, комплексов «Умный дом», охранных устройств, смартфонов, роботов, комплексов управления дорожным движением и многие другие. Язык C является одним из самых популярных языков разработки подобных систем, от которых обычно требуются высочайшая скорость выполнения и низкое потребление памяти. Например, автомобильная антиблокировочная система должна немедленно реагировать на замедление скорости вращения колес, чтобы предотвратить их блокировку; игровые приставки должны работать очень быстро, чтобы обеспечить плавность мультипликации и быструю реакцию на действия игрока
Системы реального времени	Системы реального времени часто используются для выполнения «критически важных» приложений, имеющих очень жесткие требования ко времени отклика. Например, система управления движением воздушных судов должна постоянно отслеживать положение и скорость движения самолетов и сообщать эту информацию авиадиспетчерам без каких-либо задержек, чтобы они могли вовремя сообщить экипажу самолета о необходимости изменить курс для предотвращения столкновения
Системы связи	Системы связи должны быстро управлять движением огромных объемов информации, чтобы обеспечить своевременную доставку аудио- и видеопотоков адресатам

[1] Керниган Б., Ритчи Д. Язык программирования C. – 2-е изд., перераб. и доп. – ISBN: 978-5-8459-0891-9. – М.: Вильямс, 2008. – *Прим. перев.*

Стандартизация

Быстрое распространение поддержки языка С на различные аппаратные и программные платформы способствовало появлению многочисленных особенностей, похожих, но часто несовместимых. Эти различия были серьезной проблемой для программистов, кому требовалось создавать программы для нескольких платформ сразу. Со всей очевидностью встала проблема стандартизации С. В 1983 году под эгидой Американского национального комитета по стандартизации вычислительной техники и обработки информации (American National Standards Committee on Computers and Information Processing – X3) был создан технический комитет X3J11 с целью «выработать однозначное, аппаратно-независимое определение языка». В 1989 году стандарт, получивший название ANSI X3.159-1989, был одобрен сначала в Соединенных Штатах **национальным институтом стандартов США** (**American National Standards Institute, ANSI**), а затем и **международной организацией по стандартизации** (**International Standards Organization, ISO**). Мы называем его просто «Стандарт С». Этот стандарт был дополнен в 1999 году – он получил название INCITS/ISO/IEC 9899-1999, но чаще называется просто C99. Копию текста стандарта можно заказать в национальным институте стандартов США (`www.ansi.org`) по адресу: `webstore.ansi.org/ansidocstore`.

Новый стандарт С

Мы также познакомим вас с новым стандартом С (называется C11), который был одобрен к моменту выхода книги. Новый стандарт расширяет возможности языка С. Не все современные компиляторы С поддерживают эти новые возможности. И даже те, которые обладают такой поддержкой, реализуют лишь подмножество новых особенностей. Мы добавили в книгу дополнительные разделы, где описываются новые особенности языка, поддерживаемые наиболее распространенными компиляторами.

Поскольку язык С является аппаратно-независимым, программы, написанные на нем, часто могут выполняться в самых разных системах после небольших переделок или вообще без них.

1.3 Стандартная библиотека

Программы на языке С состоят из функций. Вы можете сами написать все функции, используемые программой, но большинство программистов предпочитает пользоваться богатой коллекцией уже имеющихся функций, входящих в состав **стандартной библиотеки языка С**. То есть обучение программированию на языке С фактически можно разделить на две части – обучение самому языку С и обучение особенностям использования функций из стандартной библиотеки. На протяжении книги мы рассмот-

рим множество этих функций. Программистам, желающим глубже изучить библиотечные функции, их реализацию и особенности использования в переносимом коде, рекомендуется обратиться к книге П. Дж. Плаугера (P. J. Plauger) «The Standard C Library». Посетите также веб-сайт с документацией по стандартной библиотеке C:

```
http://ru.cppreference.com/w/c
```

Язык C поощряет создание программ методом *конструирования из строительных блоков*. Старайтесь не изобретать колесо. Используйте уже существующие компоненты. Это называется **повторным использованием программного обеспечения**. При программировании на C обычно используются следующие строительные блоки:

- функции из стандартной библиотеки;
- функции, созданные вами;
- функции, созданные другими программистами (пользующимися вашим доверием).

Преимущество создания собственных функций – в том, что вы точно знаете, как они действуют. Исходный код функций остается доступным. Недостаток – дополнительные затраты времени и сил на проектирование, разработку и отладку новых функций.

Применение функций из стандартной библиотеки вместо своих собственных может повысить производительность программы, потому что эти функции написаны с прицелом на достижение максимальной производительности.

Применение функций из стандартной библиотеки вместо своих собственных может повысить переносимость программы, потому что эти функции написаны с прицелом на достижение максимальной переносимости.

1.4 C++ и другие C-подобные языки

Язык C++ был создан Бьерном Страуструпом (Bjarne Stroustrup) из Bell Laboratories. Своими корнями он уходит в язык C, добавляя к нему множество новых особенностей. Самой важной его особенностью является поддержка **объектно-ориентированного программирования**. **Объекты** являются основными **компонентами** повторно используемого программного обеспечения, моделирующими сущности реального мира. В табл. 1.2 перечислено несколько других C-подобных языков программирования, пользующихся большой популярностью.

Таблица 1.2 | Популярные C-подобные языки программирования

Приложение	Описание
Objective-C	Objective-C – это объектно-ориентированный C-подобный язык. Был создан в начале 1980-х и позднее приобретен компанией NeXT, которая, в свою очередь, была приобретена компанией Apple. Он стал основным языком программирования для операционной системы Mac OS X и всех устройств, действующих под управлением iOS (таких как iPod, iPhone и iPad)
Visual C#	Тремя основными объектно-ориентированными языками программирования в корпорации Microsoft являются Visual Basic, Visual C++ (основан на C++) и C# (основан на C++ и Java и разрабатывался для обеспечения интеграции приложений с веб)
Java	В 1991 году компания Sun Microsystems основала исследовательский проект, в результате которого появился C++-подобный объектно-ориентированный язык программирования Java. Основной целью Java является поддержка возможности писать программы, способные выполняться на самых разных компьютерах и микропроцессорных устройствах. Для этого даже придумали термин: «написанное один раз выполняется где угодно». Язык Java используется для создания масштабируемых приложений уровня предприятия, расширения функциональных возможностей веб-серверов (компьютеров, возвращающих содержимое для отображения в наших веб-браузерах), реализации приложений для бытовых устройств (смартфонов, телевизионных приставок и др.) и для многих других целей
PHP	PHP – объектно-ориентированный, открытый (раздел 1.7) язык сценариев, основанный на C и поддерживаемый сообществом пользователей и разработчиков – используется на многих веб-сайтах, включая Wikipedia и Facebook. PHP является платформонезависимым языком – существуют его реализации для всех основных операционных систем, таких как UNIX, Linux, Mac и Windows. PHP также поддерживает множество баз данных, включая открытую MySQL. В числе других языков с аналогичной концепцией можно назвать Perl и Python
JavaScript	Язык JavaScript был разработан в компании Netscape и получил широчайшее распространение как язык сценариев. Основная область его применения – добавление программируемости в веб-страницы, например анимационных эффектов и элементов интерактивности. Поддерживается всеми основными веб-браузерами

1.5 Типичная среда разработки приложений на языке C

Системы на языке C обычно состоят из нескольких частей: среды разработки программ, языка и стандартной библиотеки. Далее в этом разделе рассматривается типичная среда разработки на языке C, изображенная на рис. 1.1.

Обычно программы на языке C проходят шесть этапов (рис. 1.1): **редактирование**, **препроцессинг**, **компиляция**, **компоновка**, **загрузка** и **выполнение**. В этом разделе мы сосредоточимся на типичной Linux-системе, основанной на языке C.

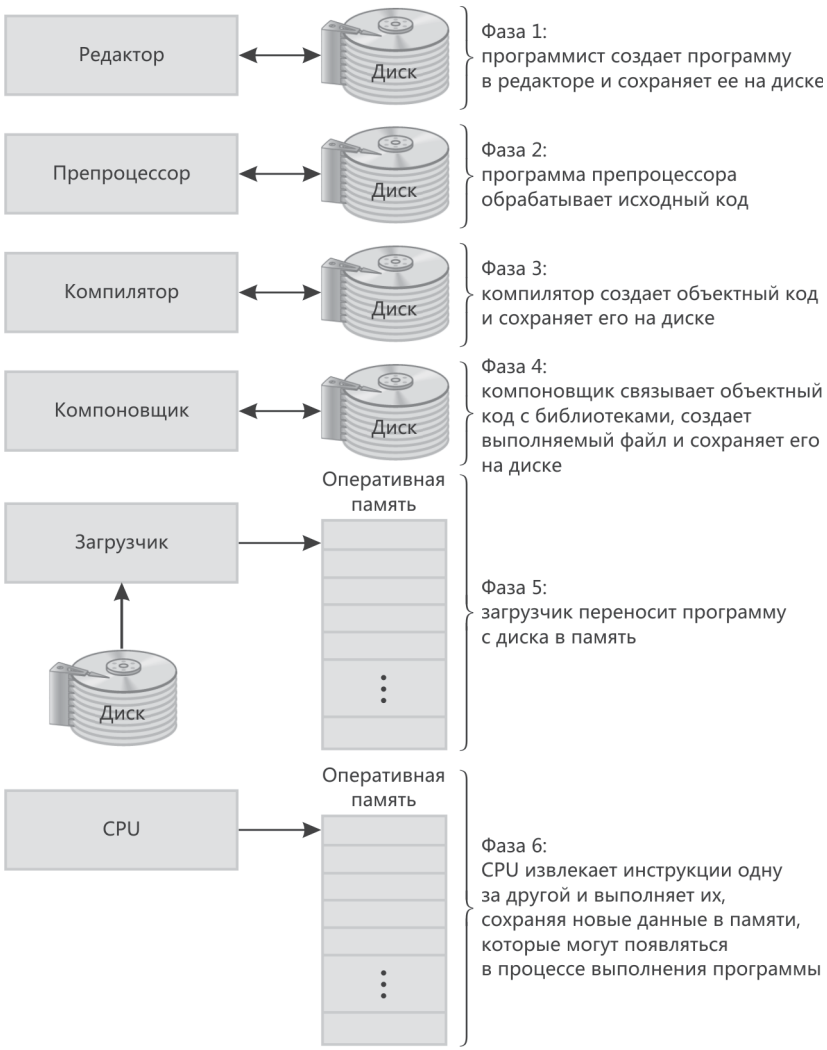


Рис. 1.1 | Типичная среда разработки приложений на языке C

1.5.1 Фаза 1: создание программы

Фаза 1 заключается в редактировании исходного кода программы. Эта фаза протекает в **программе-редакторе**. Двумя наиболее широко используемыми редакторами в Linux являются `vi` и `emacs`. Интегрированные среды

разработки программ на языке C/C++, такие как Eclipse и Microsoft Visual Studio, имеют встроенные редакторы. В течение этой фазы вы вводите исходный код программы с помощью редактора, корректируете его при необходимости, затем сохраняете на устройстве постоянного хранения, таком как жесткий диск. Файлы с исходным кодом программ на языке C должны иметь имена с расширением `.c`.

1.5.2 Фазы 2 и 3: препроцессинг и компиляция программы

В фазе 2 вы даете команду **скомпилировать** программу. Компилятор транслирует исходный код программы в машинный код (также называется объектным кодом). В системах на языке C перед фазой трансляции автоматически запускается **программа-препроцессор**. **Препроцессор языка C** выполняет специальные команды, называемые **директивами препроцессора**, определяющие операции, которые должны быть выполнены до компиляции. Обычно такими операциями являются подключение других файлов к компилируемому файлу и замена некоторых участков текста. Большинство распространенных директив препроцессора будут обсуждаться в первых главах; а в главе 13 мы более подробно рассмотрим возможности препроцессора.

В фазе 3 компилятор транслирует программу на языке C в машинный код. Если он не сможет распознать какую-либо инструкцию, генерируется **синтаксическая ошибка**, так как это считается нарушением правил языка. Чтобы помочь найти и исправить ошибочную инструкцию, компилятор выводит сообщение. Стандарт C не определяет конкретный текст сообщений об ошибках, выводимых компилятором, поэтому то, что вы увидите в своей системе, может отличаться от того, что выводится в других системах. Синтаксические ошибки также называют **ошибками компиляции**, или **ошибками времени компиляции**.

1.5.3 Фаза 4: компоновка

Следующая фаза – **компоновка**. Обычно программы на языке C содержат ссылки на функции, находящиеся где-то в другом месте, таком как стандартные библиотеки или частные библиотеки группы программистов, работающих над совместным проектом. Объектный код, производимый компилятором, чаще всего содержит «дырки», зарезервированные под отсутствующие части. **Компоновщик** связывает объектный код с кодом отсутствующих функций и генерирует **выполняемый образ** (без «дырок»). В типичной системе Linux компиляция и компоновка программ на языке C выполняются командой **`gcc`** (GNU C compiler[1]). Чтобы скомпилировать

[1] Точнее, GNU Compiler Collection – коллекция компиляторов GNU, в которой компилятор языка C – лишь один из многих. – *Прим. перев.*

и скомпоновать программу, хранящуюся в файле `welcome.c`, достаточно ввести команду

```
gcc welcome.c
```

в командной строке Linux и нажать клавишу *Enter* (или *Return*). [Имейте в виду, что команды в Linux чувствительны к регистру символов; все символы в команде `gcc` должны быть символами нижнего регистра, а регистр символов, составляющих имя файла в командной строке, должен соответствовать оригинальному имени файла.] Если программа будет скомпилирована и скомпонована без ошибок, будет создан файл `a.out`. Это – выполняемый образ программы `welcome.c`.

1.5.4 Фаза 5: загрузка

Следующая фаза – загрузка. Прежде чем программа будет запущена, ее необходимо **загрузить** в оперативную память. Эту операцию выполняет **загрузчик**, который читает выполняемый образ с диска и копирует его в оперативную память. Также загрузчик производит загрузку необходимых разделяемых библиотек.

1.5.5 Фаза 6: выполнение

Наконец, компьютер под управлением своего процессора (CPU) выполняет программу инструкция за инструкцией. Чтобы загрузить и выполнить программу в системе Linux, введите в командной строке `./a.out` и нажмите клавишу *Enter*.

1.5.6 Потоки стандартного ввода, стандартного вывода и стандартного вывода ошибок

Большинство программ на языке C вводит и/или выводит некоторые данные. Некоторые функции принимают исходные данные из **`stdin`** (**standard input stream – поток стандартного ввода**), каковым обычно является клавиатура, но поток `stdin` может быть подключен к другому потоку. Данные часто выводятся в **`stdout`** (**standard output stream – поток стандартного вывода**), каковым обычно является экран компьютера, но поток `stdout` может быть подключен к другому устройству. Когда мы говорим, что «программа вывела результаты», мы обычно подразумеваем отображение результатов на экране. Данные могут выводиться также в такие устройства, как диски и принтеры. Кроме того, существует еще **поток стандартного вывода ошибок** (**standard error stream**), который имеет имя **`stderr`**. Поток `stderr` (обычно подключенный к экрану) используется для отображения сообщений об ошибках. На практике часто используется прием перенаправления выводимых данных (то есть потока `stdout`) в устройство, отличное от экрана, а поток `stderr` оставляют подключенным к экрану, чтобы пользователь мог незамедлительно получать сообщения об ошибках.

1.6 Пробное приложение на языке C для Windows, Linux и Mac OS X

В этом разделе вы запустите и опробуете свое первое приложение на языке C. Это приложение – игра «угадай число», которая выбирает случайное число в диапазоне от 1 до 1000 и предлагает вам угадать его. Если ответ правильный – игра заканчивается. Если ответ неправильный, приложение сообщает, что ваше число больше или меньше загаданного. Ограничений на количество попыток нет, но вы должны стараться угадать число не более чем за 10 попыток. В основе этой игры лежит один интересный алгоритм – в разделе 6.8 «Поиск в массивах» вы познакомитесь с приемом поиска методом *дихотомии* (половинного деления).

Обычно подобные приложения выбирают *случайные* числа. Но в данном случае программа загадывает одну и ту же последовательность при каждом ее запуске (впрочем, такое ее поведение может зависеть от компилятора), поэтому вы можете запомнить последовательность загадываемых чисел, которая приводится ниже в данном разделе, и получить те же результаты.

Мы будем демонстрировать работу приложения в командной строке Windows, в командной оболочке Linux и в окне терминала в Mac OS X. Приложение действует совершенно одинаково на всех трех платформах. Поиграв с приложением на своей платформе, вы сможете опробовать усовершенствованную версию игры, которую можно найти в папке `randomized_version` в загружаемом архиве с примерами.

Скомпилировать и запустить программу можно в любой из множества сред разработки, таких как GNU C, Dev C++, Microsoft Visual C++, CodeLite, NetBeans, Eclipse, Xcode и др. Большинство сред разработки на языке C++ способны компилировать программы на обоих языках, C и C++.

Далее описываются шаги, выполнив которые, вы сможете запустить приложение и попытаться угадать правильное число. Элементы и функции, которые вы увидите в этом приложении, являются типичными конструкциями, с которыми вы будете знакомиться в этой книге, обучаясь программированию. Названия элементов интерфейса, отображаемых на экране, мы будем выделять рубленым шрифтом (например, **Command Prompt** (**Командная строка**)), чтобы их можно было отличить от элементов, не связанных с интерфейсом. Моноширинным шрифтом мы будем выделять имена файлов, текст, отображаемый приложением на экране, и значения, которые вводятся с клавиатуры (например, `GuessNumber` или `500`). Обратите также внимание, что **определения ключевых терминов** будут выделяться жирным шрифтом. При создании скриншотов в Windows мы изменили цвет фона окна **Command Prompt** (**Командная строка**), чтобы сделать его более читабельным. Чтобы изменить настройки цвета в приложении **Command Prompt** (**Командная строка**), запустите его, выбрав пункт меню **Start** (**Пуск**) > **All Programs**

(**Все программы**) > **Accessories** (**Стандартные**) > **Command Prompt** (**Командная строка**), щелкните правой кнопкой мыши на заголовке окна и в контекстном меню выберите пункт **Properties** (**Свойства**). В появившемся диалоге **«Command Prompt» Properties** (**Свойства «Командная строка»**) перейдите на вкладку **Colors** (**Цвета**) и выберите желаемый цвет текста и фона.

1.6.1 Запуск приложения из командной строки в Windows

1. ***Проверьте все необходимые настройки***. Обязательно прочитайте раздел «Before You Begin» на сайте `www.deitel.com/books/cfp/` и убедитесь, что без ошибок скопировали исходные тексты примеров на свой компьютер.
2. ***Перейдите в каталог с исходными текстами приложения***. Откройте окно **Command Prompt** (**Командная строка**). Перейдите в каталог с исходными текстами приложения **GuessNumber**: введите команду `cd C:\examples\ch01\GuessNumber\Windows`, затем нажмите клавишу *Enter* (рис. 1.2). Команда `cd` используется для смены каталога.

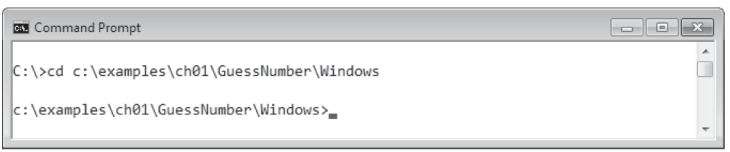


Рис. 1.2 | Откройте окно **Command Prompt** (**Командная строка**) и перейдите в каталог приложения

3. ***Запустите приложение GuessNumber***. Теперь, находясь в каталоге с исходными текстами приложения **GuessNumber**, введите команду `GuessNumber` (рис. 1.3) и нажмите клавишу *Enter*. [Обратите внимание: в действительности выполняемый файл программы имеет имя `GuessNumber.exe`, но операционная система Windows автоматически попробует добавить расширение `.exe`, если оно не указано.]

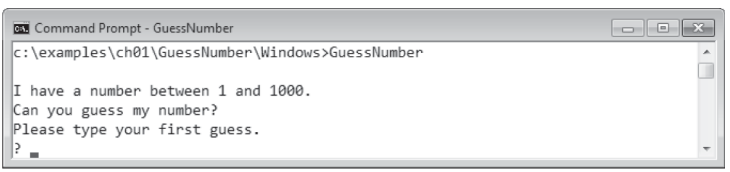


Рис. 1.3 | Запустите приложение GuessNumber

4. ***Введите первое число***. Приложение выведет приглашение к вводу `"Please type your first guess."` (Пожалуйста, введите первое число) и знак вопроса (?) в следующей строке (рис. 1.3). Введите число `500` (рис. 1.4).

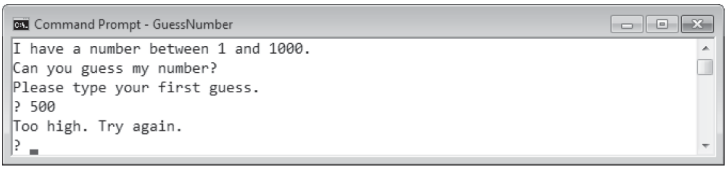


Рис. 1.4 | Введите первое число

5. ***Введите следующее число***. Приложение выведет сообщение `"Too high. Try again."` (Слишком большое, попробуйте еще), подсказывая, что введенное вами число больше загаданного. Теперь вы должны ввести число меньше предыдущего. В строке приглашения к вводу введите `250` (рис. 1.5). Приложение снова сообщит `"Too high. Try again."`, потому что введенное число также больше загаданного.

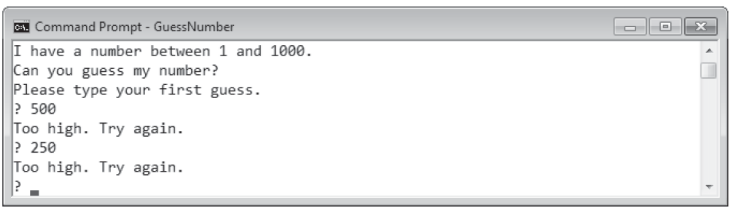


Рис. 1.5 | Введите второе число и получите ответ

6. ***Выполните дополнительные попытки***. Продолжайте игру, вводя числа, пока не угадаете загаданное. Когда это произойдет, приложение выведет `"Excellent! You guessed the number!"` (Отлично! Вы угадали число!) (рис. 1.6).
7. ***Сыграйте еще раз или завершите приложение***. Когда вы угадаете число, приложение предложит вам сыграть еще раз (рис. 1.7). Если ввести `1` в ответ на вопрос, приложение загадает новое число и выведет сообщение `"Please type your first guess."` со знаком вопроса в следующей строке (рис. 1.7), после чего вы сможете выполнить первую попытку в новом сеансе игры. Ввод числа `2` завершит приложение (рис. 1.8). Каждый раз, когда вы будете запускать приложение

заново (то есть с шага 3), оно будет загадывать те же числа и в той же последовательности.

8. ***Закройте окно Command Prompt (Командная строка).***

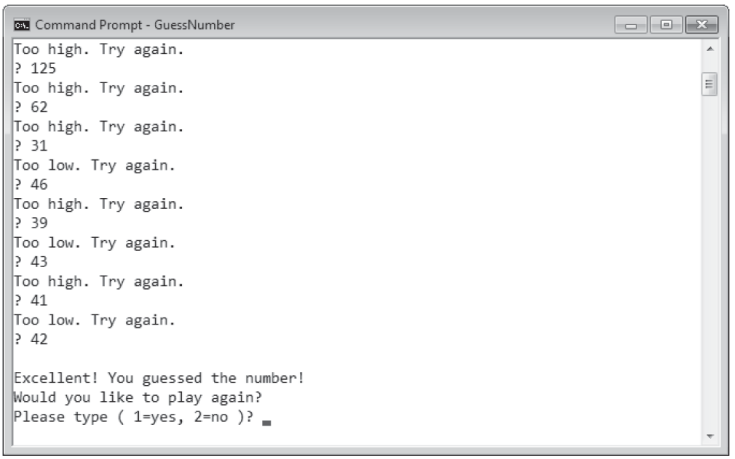


Рис. 1.6 | Продолжайте попытки, пока не угадаете число

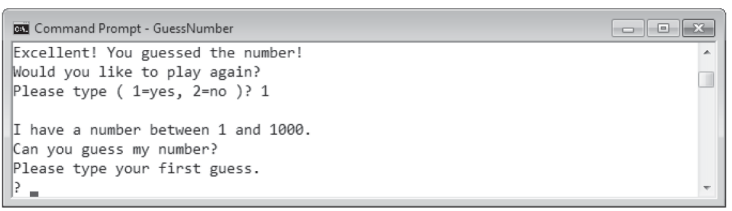


Рис. 1.7 | Сыграйте еще раз

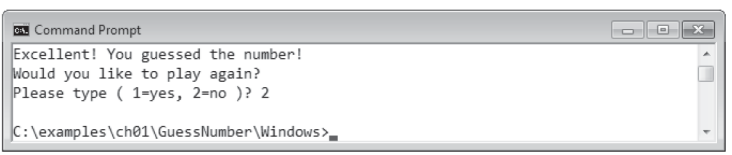


Рис. 1.8 | Покиньте игру

1.6.2 Запуск приложения в Linux

Мы полагаем, вы уже знаете, как скопировать примеры в свой домашний каталог. На скриншотах в этом разделе мы будем выделять ввод пользователя жирным шрифтом. В нашей системе строка приглашения к вводу начинается с символа тильды (~), представляющего домашний каталог, и заканчивается знаком доллара (`$`). В других системах Linux строка приглашения к вводу может отличаться.

1. ***Проверьте все необходимые настройки***. Обязательно прочитайте раздел «Before You Begin» на сайте `www.deitel.com/books/cfp/` и убедитесь, что без ошибок скопировали исходные тексты примеров на свой компьютер.
2. ***Перейдите в каталог с исходными текстами приложения***. В командной оболочке Linux перейдите в каталог с исходными текстами приложения **GuessNumber** (рис. 1.9), введя команду:

```
cd examples/ch01/GuessNumber/GNU
```

Затем нажмите клавишу *Enter*. Команда `cd` используется для смены каталога.

```
~$ cd examples/ch01/GuessNumber/GNU
~/examples/ch01/GuessNumber/GNU$
```

Рис. 1.9 | Перейдите в каталог приложения

3. ***Скомпилируйте приложение GuessNumber***. Чтобы запустить приложение, его сначала необходимо скомпилировать (рис. 1.10), введя команду

```
gcc GuessNumber.c -o GuessNumber
```

Она скомпилирует исходный код и создаст выполняемый файл `GuessNumber`.

```
~/examples/ch01/GuessNumber/GNU$ gcc GuessNumber.c -o GuessNumber
~/examples/ch01/GuessNumber/GNU$
```

Рис. 1.10 | Скомпилируйте приложение GuessNumber с помощью команды gcc

4. ***Запустите приложение GuessNumber***. Чтобы запустить выполняемый файл `GuessNumber`, введите команду `./GuessNumber` и нажмите клавишу *Enter* (рис. 1.11).
5. ***Введите первое число***. Приложение выведет приглашение к вводу `"Please type your first guess."` (Пожалуйста, введите первое число) и знак вопроса (`?`) в следующей строке (рис. 1.11). Введите число `500` (рис. 1.12).

```
~/examples/ch01/GuessNumber/GNU$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Рис. 1.11 | Запустите приложение GuessNumber

```
~/examples/ch01/GuessNumber/GNU$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Рис. 1.12 | Введите первое число

6. ***Введите следующее число***. Приложение выведет сообщение `"Too high. Try again."` (Слишком большое, попробуйте еще), подсказывая, что введенное вами число больше загаданного (рис. 1.12). В строке приглашения к вводу введите `250` (рис. 1.13). На этот раз приложение сообщит `"Too low. Try again."` (Слишком маленькое, попробуйте еще), потому что введенное число оказалось меньше загаданного.

```
~/examples/ch01/GuessNumber/GNU$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 250
Too low. Try again.
?
```

Рис. 1.13 | Введите второе число и получите ответ

7. ***Выполните дополнительные попытки***. Продолжайте игру (рис. 1.14), вводя числа, пока не угадаете загаданное. Когда это произойдет, приложение выведет `"Excellent! You guessed the number!"` (Отлично! Вы угадали число!) (рис. 1.14).

```
Too low. Try again.
? 375
Too low. Try again.
? 437
Too high. Try again.
? 406
Too high. Try again.
? 391
Too high. Try again.
? 383
Too low. Try again.
? 387
Too high. Try again.
? 385
Too high. Try again.
? 384

Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )?
```

Рис. 1.14 | Продолжайте попытки, пока не угадаете число

8. ***Сыграйте еще раз или завершите приложение***. Когда вы угадаете число, приложение предложит вам сыграть еще раз. Если ввести `1` в ответ на вопрос, приложение загадает новое число и выведет сообщение `"Please type your first guess."` со знаком вопроса в следующей строке (рис. 1.15), после чего вы сможете выполнить первую попытку в новом сеансе игры. Ввод числа `2` завершит приложение (рис. 1.16). Каждый раз, когда вы будете запускать приложение заново (то есть с шага 4), оно будет загадывать те же числа и в той же последовательности.

```
Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )? 1

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Рис. 1.15 | Сыграйте еще раз

```
Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )? 2

~/examples/ch01/GuessNumber/GNU$
```

Рис. 1.16 | Покиньте игру

1.6.3 Запуск приложения в Mac OS X

На скриншотах в этом разделе мы будем выделять ввод пользователя жирным шрифтом. Для опробования приложения в этом разделе вы будете использовать программу **Terminal** (**Терминал**). Чтобы запустить ее, щелкните на ярлыке **Spotlight Search** (**Поиск приложений**) в правом верхнем углу экрана и введите `Terminal` в поле поиска. В списке **Applications** (**Приложения**) с результатами поиска выберите пункт **Terminal** (**Терминал**). Строка приглашения к вводу в окне **Terminal** (**Терминал**) имеет вид *hostName*:~ *userFolder*$ и содержит путь к вашему домашнему каталогу. Для представления домашнего каталога в скриншотах к этому разделу мы будем использовать обезличенное имя `userFolder`.

1. ***Проверьте все необходимые настройки***. Обязательно прочитайте раздел «Before You Begin» на сайте `www.deitel.com/books/cfp/` и убедитесь, что без ошибок скопировали исходные тексты примеров на свой компьютер. Далее будет предполагаться, что исходные тексты примеров находятся в вашем домашнем каталоге, в подкаталоге `Documents/examples`.
2. ***Перейдите в каталог с исходными текстами приложения***. В окне программы **Terminal** (**Терминал**) перейдите в каталог с исходными текстами приложения **GuessNumber** (рис. 1.17), введя команду:

```
cd Documents/examples/ch01/GuessNumber/GNU
```

Затем нажмите клавишу *Enter*. Команда `cd` используется для смены каталога.

```
hostName:~ userFolder$ cd Documents/examples/ch01/GuessNumber/GNU
hostName:GNU$
```

Рис. 1.17 | Перейдите в каталог приложения

3. ***Скомпилируйте приложение GuessNumber***. Чтобы запустить приложение, его сначала необходимо скомпилировать (рис. 1.18), введя команду

```
gcc GuessNumber.c -o GuessNumber
```

Она скомпилирует исходный код и создаст выполняемый файл `GuessNumber`.

```
hostName:~ userFolder$ gcc GuessNumber.c -o GuessNumber
hostName:~ userFolder$
```

Рис. 1.18 | Скомпилируйте приложение GuessNumber с помощью команды gcc

4. ***Запустите приложение GuessNumber***. Чтобы запустить выполняемый файл `GuessNumber`, введите команду `./GuessNumber` и нажмите клавишу *Enter* (рис. 1.19).

```
hostName:~ userFolder$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Рис. 1.19 | Запустите приложение GuessNumber

5. ***Введите первое число***. Приложение выведет приглашение к вводу `"Please type your first guess."` (Пожалуйста, введите первое число) и знак вопроса (`?`) в следующей строке (рис. 1.19). Введите число `500` (рис. 1.20).

```
hostName:GNU~ userFolder$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
?
```

Рис. 1.20 | Введите первое число

6. ***Введите следующее число***. Приложение выведет сообщение `"Too low. Try again."` (Слишком маленькое, попробуйте еще), подсказывая, что введенное вами число меньше загаданного (рис. 1.12). В строке приглашения к вводу введите `750` (рис. 1.21). На этот раз приложение снова сообщит `"Too low. Try again."`, потому что введенное число оказалось меньше загаданного.

```
hostName:GNU~ userFolder$ ./GuessNumber

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
? 500
Too high. Try again.
? 750
Too low. Try again.
?
```

Рис. 1.21 | Введите второе число и получите ответ

7. ***Выполните дополнительные попытки***. Продолжайте игру (рис. 1.22), вводя числа, пока не угадаете загаданное. Когда это произойдет, приложение выведет `"Excellent! You guessed the number!"` (Отлично! Вы угадали число!).

```
Too low. Try again.
? 825
Too high. Try again.
? 788
Too low. Try again.
? 806
Too low. Try again.
? 815
Too high. Try again.
? 811
Too high. Try again.
? 808

Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )?
```

Рис. 1.22 | Продолжайте попытки, пока не угадаете число

8. ***Сыграйте еще раз или завершите приложение***. Когда вы угадаете число, приложение предложит вам сыграть еще раз. Если ввести `1` в ответ на вопрос, приложение загадает новое число и выведет сообщение `"Please type your first guess."` со знаком вопроса в следующей строке (рис. 1.23), после чего вы сможете выполнить первую попытку в новом сеансе игры. Ввод числа `2` завершит приложение (рис. 1.24). Каждый раз, когда вы будете запускать приложение за-

```
Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )? 1

I have a number between 1 and 1000.
Can you guess my number?
Please type your first guess.
?
```

Рис. 1.23 | Сыграйте еще раз

```
Excellent! You guessed the number!
Would you like to play again?
Please type ( 1=yes, 2=no )? 2

~/examples/ch01/GuessNumber/GNU$
```

Рис. 1.24 | Покиньте игру

ново (то есть с шага 3), оно будет загадывать те же числа и в той же последовательности.

1.7 Операционные системы

Операционные системы – это программные комплексы, облегчающие использование компьютеров пользователям, разработчикам приложений и системным администраторам. Они предоставляют услуги, которые позволяют приложениям выполняться безопасно, эффективно и *параллельно* с другими приложениями. Программный компонент, содержащий основные элементы операционной системы, называется **ядром**. В число популярных операционных систем для настольных компьютеров входят Linux, Windows и Mac OS X. Из популярных операционных систем для мобильных устройств, таких как смартфоны и планшетные компьютеры, можно назвать Google Android, Apple iOS (для iPhone, iPad и iPod Touch), BlackBerry OS и Microsoft Windows Phone. Вы можете разрабатывать приложения на языке C для всех четырех упомянутых платформ и многих других.

1.7.1 Windows – коммерческая операционная система

В середине 1980-х корпорация Microsoft разработала **операционную систему Windows**, обладающую графическим пользовательским интерфейсом, построенным поверх DOS – чрезвычайно популярной операционной системы для персональных компьютеров, взаимодействие с которой осуществлялось посредством *ввода* команд. ОС Windows заимствовала многие концепции (такие как ярлыки, меню и окна), разработанные в Xerox PARC и получившие популярность благодаря операционным системам Apple Macintosh. Вне всяких сомнений, Windows является самой широко распространенной операционной системой, и она является также *проприетарной* (*proprietary*) системой – развивается исключительно корпорацией Microsoft.

1.7.2 Linux – открытая операционная система

Операционная система Linux является, пожалуй, самым большим достижением движения за *открытые исходные тексты*. **Открытая модель разработки программного обеспечения** (open-source software) существенно отличается от *проприетарной* модели, доминировавшей на начальном этапе развития вычислительной техники. В открытой модели разработки отдельные специалисты и целые компании *жертвуют* (*contribute*) свои знания и силы на разработку, сопровождение и дальнейшее развитие программного обеспечения в обмен на право использовать это программное обеспечение в своих целях, обычно бесплатно. Открытый код часто исследуется намного более широкой аудиторией, чем проприетарный, поэтому ошибки в нем исправляются значительно быстрее. Кроме того, открытая модель способствует более активному продвижению инноваций. Крупные компании, за-

нимающиеся разработкой больших систем уровня предприятия, такие как IBM, Oracle и многие другие, вносят значительный вклад в разработку открытой ОС Linux.

В числе наиболее заметных организаций, широко известных в сообществе открытого программного обеспечения, можно назвать Eclipse Foundation (разрабатывает интегрированную среду разработки Eclipse, помогающую программистам со всеми удобствами заниматься разработкой программного обеспечения), Mozilla Foundation (организация-создатель веб-браузера Firefox), Apache Software Foundation (организация-создатель веб-сервера Apache, используемого для разработки веб-приложений) и SourceForge (предоставляет инструменты для управления открытыми проектами – на сайтах этой организации ведется разработка сотен тысяч открытых проектов). Быстрое развитие вычислительной техники и средств связи, уменьшение их стоимости и открытая природа программного обеспечения существенно упростили и сделали более выгодным развитие бизнеса на программном обеспечении, чем даже десять лет назад. Отличным примером может служить проект Facebook, зародившийся в университетском общежитии и построенный на основе открытого программного обеспечения.

Ядро **Linux** является основой множества популярных, открытых, свободно распространяемых полноценных операционных систем. Развивается свободно организованной командой добровольцев и широко используется в серверах, персональных компьютерах и встраиваемых системах. В отличие от коммерческих операционных систем, таких как Microsoft Windows и Apple Mac OS X, исходный код ядра Linux доступен для публичного обозрения и внесения изменений, и может бесплатно загружаться и устанавливаться. Как результат пользователи Linux пользуются поддержкой большого сообщества разработчиков, активно отлаживающих и развивающих ядро, избавлены от лицензионных выплат и ограничений, и имеют возможность глубокой настройки операционной системы под конкретные нужды.

Широкому распространению Linux в качестве операционной системы для настольных компьютеров мешают самые разные факторы, такие как засилье продуктов Microsoft на рынке, небольшое количество дружественных приложений для Linux и большое разнообразие дистрибутивов, таких как Red Hat Linux, Ubuntu Linux и др. С другой стороны, Linux завоевал небывалую популярность в качестве операционной системы для серверов и встраиваемых устройств, таких как смартфоны, действующие под управлением Google Android.

1.7.3 Apple Mac OS X, Apple iOS® для устройств iPhone®, iPad® и iPod Touch®

Компания Apple была основана в 1976 году Стивом Джобсом (Steve Jobs) и Стивом Возняком (Steve Wozniak). Она быстро заняла лидирующие позиции на рынке персональных компьютеров. В 1979 году Джобс и несколько

сотрудников компании Apple посетили Xerox PARC (научно-исследовательский центр в Пало Альто) с целью познакомиться с настольным компьютером компании Xerox, оснащенным операционной системой с графическим интерфейсом. Этот графический интерфейс послужил толчком к созданию компьютера Apple Macintosh, появившегося под большую рекламную шумиху в 1984-м.

Язык программирования Objective-C, созданный Бредом Коксом (Brad Cox) и Томом Лавом (Tom Love) в компании StepStone в начале 1980-х, добавил в C возможность объектно-ориентированного программирования (ООП). Стив Джобс (Steve Jobs) покинул Apple в 1985-м и основал NeXT Inc. В 1988-м компания NeXT приобрела лицензию на Objective-C у компании StepStone и создала компилятор Objective-C и библиотеки, которые были использованы как основа для создания пользовательского интерфейса операционной системы NeXTSTEP и построителя интерфейсов Interface Builder, используемого для конструирования графических интерфейсов.

Джобс вернулся в Apple в 1996-м, когда Apple купила NeXT. Операционная система Apple Mac OS X является производной от NeXTSTEP. Проприетарная операционная система Apple iOS стала производной от Apple Mac OS X и используется в устройствах iPhone, iPad и iPod Touch.

1.7.4 Google Android

Android – самая быстроразвивающаяся операционная система для смартфонов и других мобильных устройств – основана на ядре Linux и Java. Опытные программисты на Java могут быстро начать разрабатывать программы для Android. Одно из преимуществ платформы Android для разработчиков приложений – ее открытость. Операционная система открыта и бесплатна.

Операционная система Android была разработана компанией Android, Inc., которую приобрела компания Google в 2005-м. В 2007-м был сформирован консорциум компаний Open Handset Alliance™ с целью продолжить разработку Android. Начиная с сентября 2012-го ежедневно стало активироваться более 1,3 млн смартфонов на платформе Android[1]! Объемы продаж смартфонов на Android в настоящее время превысили объемы продаж устройств iPhone в США[2]. Операционная система Android используется во множестве смартфонов, электронных книг, планшетных компьютеров, банкоматов и платежных терминалов, автомобилей, роботов, мультимедийных плееров и многих других устройств.

---

[1] `www.pcworld.com/article/261981/android_hits_1_3_million_daily_device_activations.html`.

[2] `www.pcworld.com/article/196035/android_outsells_the_iphone_no_big_surprise.html`.

2

Введение в программирование на C

В этой главе вы научитесь:

- писать простые программы на C;
- использовать простые инструкции ввода и вывода;
- использовать базовые типы данных;
- использовать арифметические операторы;
- познакомитесь с правилами предшествования арифметических операторов;
- использовать простые условные инструкции.

Краткое содержание

2.1 Введение

Язык С способствует использованию строгого структурированного подхода к проектированию программ. В этой главе вы познакомитесь с программированием на С и несколькими примерами, иллюстрирующими многие важные особенности этого языка. В главах 3 и 4 вашему вниманию будет представлено введение в структурное программирование на С. Затем мы будем использовать структурный подход на протяжении всей книги.

2.2 Простая программа на С: вывод строки текста

Начнем со знакомства с простой программой на С. Наш первый пример выводит строку текста. Исходный код программы и ее вывод во время выполнения приводятся в примере 2.1.

Пример 2.1 | Первая программа на С и результат ее выполнения

```
1 // Пример 2.1: fig02_01.c
2 // Первая программа на С.
3 #include <stdio.h>
4
5 // выполнение программы начинается с функции main
6 int main( void )
7 {
8    printf( "Welcome to C!\n" );
9 } // конец функции main
```

```
Welcome to C!
```

Комментарии

Эта программа иллюстрирует несколько важных особенностей языка С. Строки 1 и 2

```
1 // Пример 2.1: fig02_01.c
2 // Первая программа на С.
```

начинаются с пары символов `//` – это **комментарии**. Комментарии *не* заставляют компьютер выполнять какие-либо действия при выполнении программы. Комментарии *игнорируются* компилятором С, и для них *не* генерируется какой-либо машинный код. Комментарии в первых двух строках

программы просто описывают номер примера, имя файла и назначение программы.

Языком C поддерживаются также **многострочные комментарии** `/*...*/`, где все, что заключено между парами символов `/*` и `*/`, считается комментарием. Мы предпочитаем использовать комментарии `//`, потому что они короче и избавляют от типичных программных ошибок, которые возникают при использовании комментариев `/*...*/`, например когда программист забывает вставить пару символов `*/`, закрывающую комментарий.

Директива препроцессора #include

Строка 3

```
3 #include <stdio.h>
```

– это директива **препроцессора** C. Строки, начинающиеся с символа решетки `#`, обрабатываются препроцессором перед компиляцией. Строка 3 дает указание (директиву) препроцессору включить в исходный код программы содержимое **заголовочного файла стандартной библиотеки ввода/вывода** (`<stdio.h>`). Этот заголовочный файл содержит информацию, которая потребуется компилятору, когда он будет компилировать вызовы функций из стандартной библиотеки ввода/вывода, таких как `printf` (строка 8). Более подробно содержимое заголовочных файлов будет описано в главе 5.

Пустые строки и пробелы

Строка 4 – это просто пустая строка. Пустые строки, пробелы и символы табуляции можно использовать для форматирования исходных текстов программ, чтобы придать им более удобочитаемый вид. Все вместе эти символы называют **пробелами**. Пробельные символы обычно игнорируются компилятором.

Функция main

Строка 6

```
6 int main( void )
```

является частью любой программы на языке C. Круглые скобки, следующие за именем `main`, указывают, что `main` – это функция. Программы на C всегда содержат одну или более функций, одна из которых *должна* иметь имя `main`. Все программы на C начинают выполняться с функции `main`. Функции могут *возвращать* информацию. Ключевое слово `int`, слева от имени `main`, указывает, что функция `main` «возвращает» целое (integer) число. Подробнее о том, что это означает, мы поговорим, когда будем знакомиться с приемами создания собственных функций в главе 5. А пока просто добавляйте ключевое слово `int` слева от имени `main` в каждой своей программе. Функции также могут *принимать* информацию в момент вызова, в процессе выполне-

ния программы. Ключевое слово `void` в круглых скобках в данном примере означает, что функция `main` *не* принимает никакой информации. В главе 14 мы познакомимся с примером функции `main`, принимающей информацию.

Каждую функцию следует снабжать комментарием, описывающим ее назначение.

Открывающая фигурная скобка `{` начинает **тело** любой функции (строка 7). Парная ей **закрывающая фигурная скобка** `}` завершает тело функции (строка 9). Пара фигурных скобок и фрагмент программы между ними называются *блоком*. Блоки – важные элементы программ на языке C.

Инструкция вывода

Строка 8

```
printf( "Welcome to C!\n" );
```

предписывает компьютеру выполнить **действие**, а именно вывести на экран строку символов, заключенную в кавычки. Строки символов иногда называют просто **строками**, **сообщениями** или **литералами**. Вся строка, включая имя функции `printf` (символ «f» в имени обозначает «formatted» – «форматированный»), ее **аргументы** в круглых скобках и точку с запятой (`;`), называется **инструкцией**. Каждая инструкция должна заканчиваться символом точки с запятой (который также называют **ограничителем инструкции**). Когда описываемая здесь инструкция `printf` выполняется, она выводит на экран сообщение `Welcome to C!`. Обычно символы выводятся именно так, как они указаны в тексте между двумя кавычками.

Экранированные последовательности

Обратите внимание, что символы `\n` в примере выше не выводятся на экран. Обратный слэш (\) называют **символом экранирования** (или **escape-символом**). Он указывает, что `printf` должна сделать нечто необычное. Встречая символ обратного слэша в строке, компилятор извлекает следующий символ и объединяет его с обратным слэшем в **экранированную последовательность** (или escape-последовательность – escape sequence). Экранированная последовательность `\n` соответствует символу перевода строки. Когда `printf` встречает символ перевода строки, она переносит текстовый курсор в начало следующей экранной строки. Некоторые наиболее часто используемые экранированные последовательности перечислены в табл. 2.1.

Так как в строках символ обратного слэша приобретает специальное значение, то есть компилятор распознает его как экранирующий символ, чтобы вывести на экран один символ обратного слэша, необходимо вставить в строку два обратных слэша (\\). Вывод двойной кавычки также представляет проблему, потому что двойные кавычки в тексте программы играют роль ограничителей строк – эти кавычки не выводятся на экран. Используя экра-

Таблица 2.1 | Некоторые экранированные последовательности

Экранированная последовательность	Описание
\n	Перевод строки. Текстовый курсор перемещается в начало следующей строки
\t	Горизонтальная табуляция. Текстовый курсор перемещается к следующей позиции табуляции
\a	Сигнал. Производит звуковой или визуальный сигнал, не изменяя текущего положения текстового курсора
\\	Обратный слэш. Вставляет в строку символ обратного слэша
\"	Двойная кавычка. Вставляет в строку символ двойной кавычки

нированную последовательность `\"` в строке, которая выводится функцией `printf`, мы можем потребовать от `printf` вывести двойную кавычку. Правая скобка `}` (строка 9) отмечает конец функции `main`.

Добавляйте комментарии ко всем фигурным скобкам `}`, закрывающим функции, в том числе и функцию `main`.

Выше было сказано, что `printf` заставляет компьютер выполнить **действие**. В процессе выполнения любая программа производит некоторые действия и принимает **решения**. Подробнее о принятии решений мы поговорим в разделе 2.5. А в главе 3 детально будет обсуждаться модель программирования **действие/решение**.

Компоновщик и выполняемые файлы

Функции из стандартной библиотеки, такие как `printf` и `scanf`, *не* являются частью языка C. Например, компилятор *не обнаружит* опечатку в имени `printf` или `scanf`. Когда компилятор компилирует инструкцию `printf`, он просто резервирует место в объектном коде для машинных инструкций «вызова» библиотечной функции. Но компилятор *не* знает, где находятся библиотечные функции, – этой информацией владеет *компоновщик* (*linker*). Когда запускается компоновщик, он отыскивает библиотечные функции и вставляет соответствующие инструкции вызова в объектный код. После обработки компоновщиком объектная программа становится готовой к выполнению. Именно поэтому скомпонованный файл программы называется **выполняемым** (executable). Если имя функции будет записано с ошибкой, *компоновщик* обнаружит ее, потому что не сможет найти в библиотеках функцию с таким именем.

Смещайте все тело функции на один отступ вправо (мы рекомендуем использовать отступы шириной в три пробела). Отступы будут визуально подчеркивать функциональную структуру программы и сделают ее проще для чтения.

Определите для себя соглашение о наиболее предпочтительной величине отступов и неуклонно следуйте этому соглашению. Для создания отступов можно использовать клавишу табуляции, но в разных редакторах ширина табуляции может быть настроена по-разному.

Использование нескольких инструкций printf

Функция `printf` может вывести текст `Welcome to C!` несколькими разными способами. Например, программа в примере 2.2 выводит тот же текст, что и программа из примера 2.1, но использует для этого несколько инструкций `printf`, каждая из которых продолжает вывод с того места, где его прекратила предыдущая инструкция `printf`. Первая инструкция `printf` (строка 8) выводит `Welcome` и пробел, вторая инструкция `printf` (строка 9) начинает вывод в той же строке, сразу за пробелом.

Пример 2.2 | Вывод одной строки двумя инструкциями printf

```
 1 // Пример 2.2: fig02_03.c
 2 // Вывод одной строки двумя инструкциями printf.
 3 #include <stdio.h>
 4
 5 // Выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    printf( "Welcome " );
 9    printf( "to C!\n" );
10 } // конец функции main
```

```
Welcome to C!
```

Одна инструкция `printf` может вывести *несколько* строк, используя символы перевода строки, как показано в примере 2.3. Каждый раз, когда встречается экранированная последовательность `\n` (перевод строки), вывод продолжается с начала новой строки.

Пример 2.3 | Вывод нескольких строк одной инструкцией printf

```
1 // Пример 2.3: fig02_04.c
2 // Вывод нескольких строк одной инструкцией printf.
3 #include <stdio.h>
4
5 // Выполнение программы начинается с функции main
6 int main( void )
7 {
8    printf( "Welcome\nto\nC!\n" );
9 } // конец функции main
```

```
Welcome
to
C!
```

2.3 Еще одна простая программа на C: сложение двух целых чисел

Следующая наша программа использует функцию `scanf` из стандартной библиотеки, чтобы получить два целых числа, введенных пользователем с клавиатуры, вычисляет сумму этих чисел и выводит результат с помощью `printf`. Программа и ее вывод приводятся в примере 2.4. [В листинге, иллюстрирующем сеанс взаимодействия с программой, числа, введенные пользователем, мы выделили **жирным**.]

Пример 2.4 | Программа сложения

```
 1 // Пример 2.4: fig02_05.c
 2 // Программа сложения.
 3 #include <stdio.h>
 4
 5 // Выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    int integer1; // первое число, введенное пользователем
 9    int integer2; // второе число, введенное пользователем
10    int sum;      // переменная для хранения суммы
11
12    printf( "Enter first integer\n" );    // вывести текст приглашения
                                            // к вводу
13    scanf( "%d", &integer1 );             // прочитать целое число
14
15    printf( "Enter second integer\n" );   // вывести текст приглашения
                                            // к вводу
16    scanf( "%d", &integer2 );             // прочитать целое число
17
18    sum = integer1 + integer2;            // вычислить и сохранить сумму
19
20    printf( "Sum is %d\n", sum );         // вывести сумму
21 } // конец функции main
```

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Комментарий в строке сообщает назначение программы. Как уже отмечалось выше, выполнение любой программы на языке C начинается с функции `main`. Открывающая фигурная скобка `{` (строка 7) обозначает начало тела функции `main`, а парная ей закрывающая фигурная скобка `}` (строка 21) – конец функции `main`.

Переменные и определения переменных

Строки 8–10

```
 8    int integer1; // первое число, введенное пользователем
 9    int integer2; // второе число, введенное пользователем
10    int sum;      // переменная для хранения суммы
```

– это **определения**. Имена `integer1`, `integer2` и `sum` – это имена переменных – участки памяти, где можно хранить значения, используемые программой. Эти определения указывают, что переменные `integer1`, `integer2` и `sum` имеют тип `int`, то есть они хранят **целочисленные значения**, такие как 7, –11, 0, 31914 и им подобные.

Перед использованием любой переменной необходимо определить ее имя и тип. Читатели, имеющие опыт использования компилятора Microsoft Visual C++, должны обратить внимание, что мы поместили определения переменных сразу после открывающей фигурной скобки, отмечающей начало тела функции `main`. Стандарт языка C позволяет помещать определения переменных в *любом месте*, непосредственно перед их использованием в коде. Некоторые компиляторы, такие как GNU `gcc`, поддерживают подобную возможность. Более подробно эта проблема будет рассматриваться в следующих главах.

Определения переменных в примере выше можно объединить в одну инструкцию, как показано ниже:

```
int integer1, integer2, sum;
```

но такой способ осложняет комментирование объявлений переменных, как показано в строках 8–10.

Идентификаторы и чувствительность к регистру

Именем переменной в языке C может быть любой допустимый **идентификатор**. Идентификатор – это последовательность символов – букв, цифр и символов подчеркивания (_). При этом идентификаторы *не* могут начинаться с цифры. Идентификаторы в C чувствительны к регистру символов – буквы верхнего и нижнего регистров считаются *разными*, то есть `a1` и `A1` – это *разные* идентификаторы.

Старайтесь избегать идентификаторов, начинающихся с символа подчеркивания (_), чтобы предотвратить конфликты с идентификаторами, генерируемыми компилятором и объявленными в стандартной библиотеке.

Желательно, чтобы первым символом в идентификаторе, служащем именем простой переменной, была буква нижнего регистра. Далее в книге мы будем придавать особое значение идентификаторам, в которых используются только буквы верхнего регистра.

Использование имен переменных, состоящих из нескольких слов, может улучшить читабельность программы. Слова в именах можно отделять символом подчеркивания, например `total_commissions`*, или, если слова записываются слитно, каждое последующее слово, кроме первого, можно начинать с буквы верхнего регистра, например* `totalCommissions`*. Последний способ считается наиболее предпочтительным.*

Синтаксические ошибки

В главе 1 мы уже упоминали синтаксические ошибки. Напомню, что компилятор Microsoft Visual C++ требует, чтобы определения переменных помещались сразу *за* фигурной скобкой, открывающей тело функции, и *перед* любыми выполняемыми инструкциями. То есть если в программе из примера 2.4 объявление переменной `integer1` поместить *после* первой инструкции `printf`, это вызовет синтаксическую ошибку.

Вставка объявления переменной между выполняемыми инструкциями вызовет в Microsoft Visual C++ синтаксическую ошибку.

Сообщения с приглашением к вводу

Строка 12

```
12    printf( "Enter first integer\n" );  // вывести текст приглашения
                                          // к вводу
```

выводит текст `"Enter first integer"` (Введите первое целое число) и устанавливает текстовый курсор в начало следующей строки. Это сообщение называется **приглашением к вводу** (prompt), потому что оно предлагает пользователю выполнить вполне определенное действие.

Функция scanf и форматированный ввод

Следующая инструкция

```
13    scanf( "%d", &integer1 );           // прочитать целое число
```

вызывает функцию `scanf` (символ «f» в имени обозначает «formatted» – «форматированный»), чтобы получить число, введенное пользователем. Она читает символы со *стандартного ввода*, каковым обычно является клавиатура. В данном случае `scanf` принимает два аргумента, `"%d"` и `&integer1`. Первый – это **строка формата**, определяющая *тип* данных, которые должен ввести пользователь. **Спецификатор преобразования** `%d` указывает, что пользователь должен ввести целое число (символ «d» обозначает «decimal integer» – «десятичное целое»). Символ `%` в данном контексте интерпретируется функцией `scanf` (и функцией `printf`) как специальный символ, с которого начинаются спецификаторы преобразований. Второй аргумент функции `scanf` начинается с символа амперсанда (&) – называется **операто-**

ром взятия адреса, – за которым следует имя переменной. Знак амперсанда & в комбинации с именем переменной сообщает функции `scanf` местоположение (или адрес) в памяти переменной `integer1`. Когда пользователь введет число, компьютер сохранит его в переменной `integer1`. Применение знака амперсанда часто приводит в замешательство начинающих программистов или тех, кто уже имеет опыт программирования на других языках, не требующих использовать подобную форму записи. С этим мы разберемся позже, а пока просто запомните, что каждая переменная должна передаваться функции `scanf` со знаком амперсанда. Некоторые исключения из этого правила мы рассмотрим в главах 6 и 7. Правила использования амперсанда станут более очевидными, когда мы познакомимся с *указателями* в главе 7.

Добавляйте пробел после запятой (,), чтобы сделать текст программы более читабельным.

Когда в процессе выполнения программы компьютер встретит инструкцию `scanf`, он остановится в ожидании, пока пользователь введет число для переменной `integer1`. Пользователь должен ввести число и затем нажать клавишу *Enter*, чтобы передать число компьютеру. После этого компьютер присвоит это число, или значение, переменной `integer1`. В ответ на любые последующие обращения к переменной `integer1` в программе будет возвращаться это значение. Функции `printf` и `scanf` упрощают организацию взаимодействий между пользователем и компьютером. Поскольку подобное взаимодействие напоминает диалог, его часто называют **интерактивным взаимодействием**.

Строка 15

```
15    printf( "Enter second integer\n" );  // вывести текст приглашения
                                           // к вводу
```

выводит сообщение `"Enter second integer"` (Введите второе целое число) и устанавливает текстовый курсор в начало следующей строки. Данная инструкция `printf` также предлагает пользователю выполнить действие.

Строка 16

```
16    scanf( "%d", &integer2 );            // прочитать целое число
```

получает от пользователя значение для переменной `integer2`.

Инструкция присваивания

Инструкция присваивания в строке 18

```
18    sum = integer1 + integer2;           // вычислить и сохранить сумму
```

вычисляет сумму значений переменных `integer1` и `integer2` и присваивает результат переменной `sum`. Эта инструкция читается как «sum *получает* значение выражения `integer1 + integer2`». В большинстве случаев вычисления

выполняются в инструкциях присваивания. Оператор = и оператор + называются двухместными[1] операторами, потому что принимают *два* **операнда**. Операндами оператора + являются `integer1` и `integer2`. Операндами оператора = являются `sum` и значение выражения `integer1 + integer2`.

Добавляйте пробелы слева и справа от двухместного оператора для удобочитаемости.

Вывод с использованием строки формата

В строке 20

```
20    printf( "Sum is %d\n", sum );       // вывести сумму
```

производится вызов функции `printf`, которая выводит строку `"Sum"` (Сумма) и числовое значение переменной `sum`. В данном случае `printf` принимает два аргумента, `"Sum is %d\n"` и `sum`. Первый аргумент – это строка формата. Она содержит несколько литеральных символов и спецификатор преобразования `%d`, указывающий, что выводиться будет целое число. Второй аргумент определяет значение для вывода. Обратите внимание, что в обеих функциях, `printf` и `scanf`, используется один и тот же спецификатор, обозначающий целое число.

Вычисления в инструкциях printf

Вычисления могут также выполняться непосредственно внутри инструкций `printf`. Мы могли бы объединить две предыдущие инструкции в одну

```
printf( "Sum is %d\n", integer1 + integer2 );
```

Закрывающая фигурная скобка `}` в строке 21 отмечает конец функции `main`.

Если в вызове функции `scanf` забыть добавить амперсанд перед именем переменной, когда он должен там быть, это приведет к ошибке своевременного выполнения. Во многих системах такая забывчивость порождает ошибку «segmentation fault» (нарушение сегментирования) или «access violation» (нарушение прав доступа). Такие ошибки возникают, когда программа пытается обратиться к разделам памяти, для доступа к которым у программы недостаточно привилегий. Более точное описание причин, вызывающих подобные ошибки, приводится в главе 7.

Ошибкой также считается добавление амперсанда перед именем переменной, когда его не должно быть.

---

[1] Иногда можно встретить название *бинарный* (binary). – *Прим. перев.*

2.4 Арифметические операции в языке C

Большинство программ на языке C выполняют арифметические вычисления, используя для этого **арифметические операторы** (табл. 2.2). **Звездочка** (*) обозначает операцию *умножения*, **знак процента** (%) – оператор получения *остатка от деления*, который будет представлен ниже. В алгебре умножение *a* на *b* записывается просто как *ab*. Однако если то же самое сделать в программе на C, последовательность символов *ab* будет интерпретироваться как двухбуквенное имя переменной (или идентификатор). То есть операция умножения должна обозначаться явно, с помощью оператора *, например `a * b`. Все арифметические операторы являются *двухместными*. Например, выражение `3 + 7` содержит двухместный оператор + и два операнда, 3 и 7.

Таблица 2.2 | Арифметические операторы

Операция	Оператор	Алгебраическое выражение	Выражение на языке C
Сложение	+	$f + 7$	f + 7
Вычитание	-	$p - c$	p - c
Умножение	*	bm	b * m
Деление	/	x / y или $\frac{x}{y}$ или $x \div y$	x / y
Остаток от деления	%	$r \bmod s$	r % s

Целочисленное деление и оператор остатка от деления

При делении целых чисел возвращается целое число. Например, выражение `7 / 4` вернет `1`, а выражение `17 / 5` вернет `3`. В языке C имеется **оператор остатка от деления** `%`, который возвращает *остаток* от целочисленного деления. Оператор остатка – это целочисленный оператор, который можно использовать только с целочисленными операндами. Выражение `x % y` вернет остаток от деления `x` на `y`. То есть `7 % 4` вернет `3`, а `17 % 5` вернет `2`. Далее в книге мы обсудим множество интересных применений оператора остатка от деления.

Попытка деления на ноль в компьютерных системах обычно приводит к фатальной ошибке, то есть ошибке, вызывающей немедленное завершение программы. Нефатальные ошибки позволяют программам продолжить выполнение, но нередко приводят к ошибочным результатам.

Линейная форма записи арифметических выражений

Арифметические выражения в языке C должны записываться в **линейной форме**, чтобы упростить ее выполнение компьютером. То есть выражения, такие как «a разделить на b», должны записываться как `a/b`, чтобы все операторы и операнды находились в одной строке. Алгебраическая форма записи

$$\frac{a}{b}$$

обычно недопустима при работе с компьютерами, хотя некоторые специализированные программные пакеты поддерживают такую, более естественную форму записи сложных математических выражений.

Группировка подвыражений с помощью круглых скобок

Круглые скобки используются в выражениях на языке C с той же целью, что и в алгебраических выражениях. Например, чтобы умножить `a` на `b + c`, можно записать выражение `a * ( b + c )`.

Правила предшествования операторов

В арифметических выражениях на языке C операторы выполняются в определенном порядке, с соблюдением следующих **правил предшествования**, которые в значительной степени совпадают с правилами, используемыми в алгебре:

1. Сначала выполняются операторы в выражениях, заключенных в круглые скобки. Круглые скобки имеют «наивысший приоритет». В случае наличия в выражении **вложенных скобок**, как, например, в выражении:

```
( ( a + b ) + c )
```

   первым вычисляется оператор в самых *внутренних* скобках.
2. Далее выполняются операторы умножения, деления и остатка от деления. Если выражение содержит несколько операторов умножения, деления или остатка от деления, они вычисляются в порядке слева направо. Операторы умножения, деления или остатка от деления имеют одинаковый приоритет.
3. Далее вычисляются операторы сложения и вычитания. Если выражение содержит несколько операторов сложения и вычитания, они вычисляются в порядке слева направо. Операторы сложения и вычитания также имеют одинаковый приоритет, который ниже приоритета операторов умножения, деления и остатка от деления.
4. Последним выполняется оператор присваивания (`=`).

Правила предшествования определяют порядок вычисления выражений в языке C[1]. Когда мы говорим о порядке выполнения слева направо, мы подразумеваем **ассоциативность** операторов. Далее нам встретятся несколько операторов, имеющих ассоциативность справа налево. В табл. 2.3

[1] Для описания порядка вычисления операторов мы использовали простые выражения. Однако в более сложных выражениях возникают некоторые малозаметные проблемы, о которых мы будем рассказывать по мере их появления.

приводятся правила предшествования операторов, которые встречались нам до сих пор.

Таблица 2.3 | Предшествование арифметических операторов

Оператор(ы)	Операция	Порядок вычисления (предшествования)
()	Круглые скобки	Вычисляются первыми. Если имеются вложенные скобки, в первую очередь вычисляется выражение в самых вложенных скобках. Если имеется несколько пар круглых скобок «на одном уровне» (то есть невложенных), они вычисляются слева направо
*	Умножение	Вычисляется вторым. Если имеется несколько операторов, они вычисляются слева направо
/	Деление	
%	Остаток	
+	Сложение	Вычисляется третьим. Если имеется несколько операторов, они вычисляются слева направо
-	Вычитание	
=	Присваивание	Вычисляется последним

Примеры выражений, алгебраических и на языке C

Теперь рассмотрим несколько выражений в свете правил предшествования операторов. Каждый пример будет включать выражение в алгебраической записи и на языке C. Следующее выражение вычисляет среднее арифметическое пяти значений.

Алгебраическая форма записи: $m = \frac{a + b + c + d + e}{5}$.

```
Форма записи на языке C: m = ( a + b + c + d + e ) / 5;
```

Круглые скобки, окружающие группу операторов сложения, здесь совершенно необходимы, потому что деление имеет более высокий приоритет, чем сложение. В данном случае делиться на 5 должно все выражение `( a + b + c + d + e )`. Если скобки опустить, мы получим выражение `a + b + c + d + e / 5`, дающее совсем другой результат:

$$a + b + c + d + \frac{e}{5}$$

Следующее выражение записывается линейно:

Алгебраическая форма записи: $y = mx + b$.

```
Форма записи на языке C: y = m * x + b;
```

Здесь круглые скобки не нужны. Первым будет выполнен оператор умножения как имеющий более высокий приоритет перед оператором сложения.

Следующее выражение содержит операторы получения остатка от деления (%), умножения, деления, сложения, вычитания и присваивания:

```
Алгебраическая форма записи: y = p % q + w / x – y.
Форма записи на языке C: z = p * r % q + w / x - y;
                           ❻  ❶  ❷  ❹  ❸  ❺
```

Цифры в кружочках указывают очередность вычисления операторов в языке C. Операторы умножения, остатка от деления и деления выполняются первыми, в порядке слева направо (то есть все они являются ассоциативными слева направо), потому что они имеют более высокий приоритет, чем операторы сложения и вычитания. Затем вычисляются операторы сложения и вычитания. Они также выполняются в порядке слева направо. Последним выполняется оператор присваивания, сохраняющий результат в переменной `z`.

Не все выражения с несколькими парами круглых скобок содержат вложенные скобки. Например, следующее выражение не содержит вложенных скобок – здесь скобки находятся «на одном уровне»:

```
a * ( b + c ) + c * ( d + e )
```

Вычисление значения полинома второй степени

Чтобы вы могли полнее усвоить правила предшествования операторов, рассмотрим пример вычисления значения полинома второй степени на языке C.

```
y = a * x * x + b * x + c;
  ❻  ❶  ❷  ❹  ❸  ❺
```

Цифры в кружочках под инструкцией указывают очередность вычисления операторов. В языке C нет оператора возведения в степень, поэтому член полинома x^2 здесь представлен выражением `x * x`. Стандартная библиотека C включает в себя функцию `pow` («power» – «степень») возведения в степень, но из-за некоторых тонкостей, связанных с типами данных, которые должны передаваться функции `pow`, мы пока отложим знакомство с ней до главы 4.

Допустим, что переменные `a`, `b`, `c` и `x` в предыдущем выражении инициализированы следующими значениями: `a = 2`, `b = 3`, `c = 7` и `x = 5`. На рис. 2.1 показано, в каком порядке будут вычисляться операторы.

Как и в алгебре, в языке C допускается вставлять необязательные круглые скобки, чтобы упростить чтение выражения. Такие скобки называют **избыточными**. Например, в предыдущую инструкцию можно было бы включить круглые скобки, как показано ниже:

```
y = ( a * x * x ) + ( b * x ) + c;
```

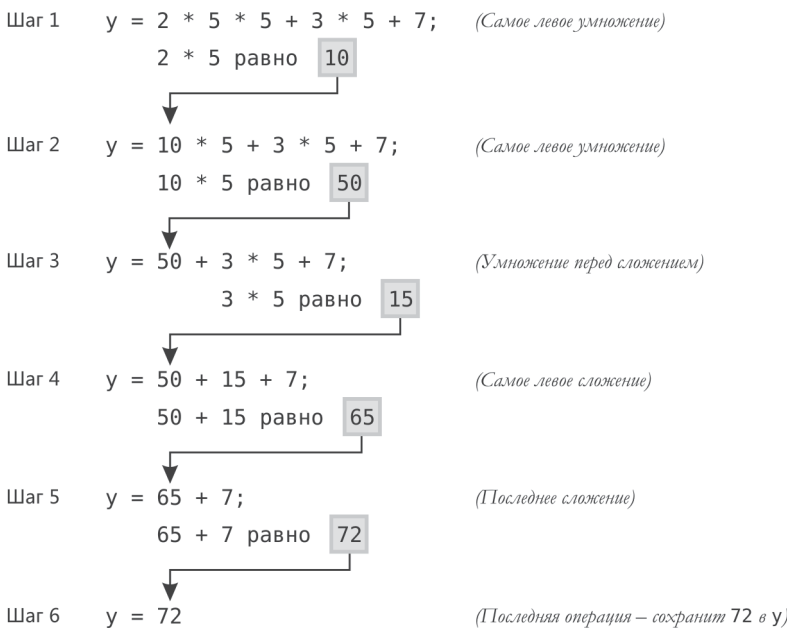


Рис. 2.1 | Порядок вычисления полинома второй степени

2.5 Принятие решений: операторы сравнения

Выполняемые инструкции либо выполняют некоторые действия (например, вычисления или ввод/вывод), либо принимают решения (несколько примеров приводятся ниже). У нас имеется возможность принимать решения в программе, например выяснить, набрал ли ученик на экзамене больше 60 баллов, и должна ли программа вывести на экран сообщение: «Поздравляем! Экзамен сдан!» В этом разделе мы познакомимся с простейшей версией **условной инструкции** `if` в языке C, позволяющей программе принимать решения, исходя из истинности или ложности **условного выражения**. Если условное выражение **истинно** (то есть требуемое условие соблюдается), выполняется тело инструкции `if`. Если условное выражение **ложно** (то есть требуемое условие не соблюдается), тело инструкции `if` не выполняется. Независимо от того, выполняется тело инструкции `if` или нет, выполнение программы будет продолжено с инструкции, следующей за условной инструкцией `if`.

Условные выражения в инструкциях `if` образуются с помощью **операторов сравнения** и **операторов отношений**, перечисленных в табл. 2.4. Все операторы отношений имеют одинаковый приоритет и ассоциативность слева направо. Оператор равенства имеет более низкий приоритет, чем операторы отношений, и также имеет ассоциативность слева направо. [Обратите внимание, что в языке C роль условного выражения может играть *любое выражение, возвращающее нулевое (ложное) или ненулевое (истинное) значение*.]

Оператор равенства == часто путают с оператором присваивания. Чтобы избежать путаницы, оператор сравнения следует читать как «двойное сравнение», а оператор присваивания – как «присвоить». Как будет показано далее, путаница между этими операторами не всегда вызывает ошибку компиляции, но может вызывать грубые логические ошибки во время выполнения.

Таблица 2.4 | Операторы равенства и отношений

Алгебраический оператор равенства или отношения	Оператор равенства или отношения в языке C	Пример условного выражения на языке C	Интерпретация условного выражения на языке C
Операторы сравнения			
=	==	`x == y`	x равно y
≠	!=	`x != y`	x не равно y
Операторы отношений			
>	>	`x > y`	x больше, чем y
<	<	`x < y`	x меньше, чем y
≥	>=	`x >= y`	x больше или равно y
≤	<=	`x <= y`	x меньше или равно y

В примере 2.5 используются шесть инструкций `if` для сравнения двух чисел, введенных пользователем. Если условное выражение какой-либо из этих инструкций вернет истинное значение, будет выполнена соответствующая инструкция `printf`. Вслед за примером показаны три сеанса работы с программой.

Пример 2.5 | Использование инструкций if, операторов отношений и операторов равенства

```
1 // Пример 2.5: fig02_10.c
2 // Использование инструкций if, операторов отношений
3 // и операторов равенства.
4 #include <stdio.h>
5
```

```
 6 // выполнение программы начинается с функции main
 7 int main( void )
 8 {
 9    int num1; // первое число, введенное пользователем
10    int num2; // второе число, введенное пользователем
11
12    printf( "Enter two integers, and I will tell you\n" );
13    printf( "the relationships they satisfy: " );
14
15    scanf( "%d%d", &num1, &num2 ); // прочитать два целых числа
16
17    if    if ( num1 == num2 ) {
18       printf( "%d is equal to %d\n", num1, num2 );
19    } // конец if
20
21    if( num1 != num2 ){
22       printf( "%d is not equal to %d\n", num1, num2 );
23    } // конец if
24
25    if( num1 < num2 ){
26       printf( "%d is less than %d\n", num1, num2 );
27    } // конец if
28
29    if( num1 > num2 ){
30       printf( "%d is greater than %d\n", num1, num2 );
31    } // конец if
32
33    if( num1 <= num2 ){
34       printf( "%d is less than or equal to %d\n", num1, num2 );
35    } // конец if
36
37    if( num1 <= num2 ){
38       printf( "%d is greater than or equal to %d\n", num1, num2 );
39    } // конец if
40 } // конец функции main
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 3 7
3 is not equal to 7
3 is less than 7
3 is less than or equal to 7
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 22 12
22 is not equal to 12
22 is greater than 12
22 is greater than or equal to 12
```

```
Enter two integers, and I will tell you
the relationships they satisfy: 7 7
7 is equal to 7
7 is less than or equal to 7
7 is greater than or equal to 7
```

Ввод двух чисел программой осуществляется с помощью функции `scanf` (строка 15). Для каждого спецификатора преобразования функции переда-

ется соответствующий аргумент, куда будет сохраняться прочтенное значение. Первый спецификатор `%d` сохранит введенное значение в переменной `num1`, а второй – в переменной `num2`.

Хотя это и допустимо, все же не следует указывать более одной инструкции в одной строке программы.

Не разделяйте спецификаторы преобразований запятыми (если они не нужны) в строке формата в вызове функции `scanf`.

Сравнение чисел

Инструкция `if` в строках 17–19

```
17    if    if ( num1 == num2 ) {
18       printf( "%d is equal to %d\n", num1, num2 );
19    } // конец if
```

проверяет значения переменных `num1` и `num2` на равенство. Если значения равны, инструкция в строке 18 выведет текст, сообщающий о равенстве значений. Если условное выражение в какой-либо из инструкций – в строках 21, 25, 29, 33 и 37 – вернет истинное значение, выполнится тело этой условной инструкции и на экран будет выведен соответствующий текст. Наличие дополнительных отступов в телах условных инструкций и пробельные строки выше и ниже каждой инструкции `if` повышают удобочитаемость программы.

Не добавляйте точку с запятой сразу за круглой скобкой, закрывающей условное выражение в инструкции `if`.

Тело каждой условной инструкции `if` начинается с открывающей фигурной скобки `{` (как, например, в строке 17). Парная ей закрывающая фигурная скобка `}` завершает тело инструкции `if` (как, например, в строке 19). Тело инструкции `if` может состоять из любого количества инструкций[1].

Длинную инструкцию можно разбить на несколько строк. В этом случае осмысленно подходите к выбору точек разбиения инструкций (например, переносите строки после запятых в списках, где значения разделяются запятыми). Разбивая инструкцию на несколько строк, все последующие ее части дополняйте отступами. Идентификаторы разбивать нельзя.

[1] Фигурные скобки можно не использовать, если тело условной инструкции состоит из единственной инструкции. Многие программисты считают хорошей практикой всегда использовать фигурные скобки. Причину этого мы разъясним в главе 3.

В табл. 2.5 перечислены операторы, представленные в этой главе, в порядке следования приоритетов от высшего к низшему. В верхней части таблицы перечисляются операторы с высшим приоритетом, в нижней – с низшим. Знак «равно» (=) также является оператором. Все эти операторы, кроме оператора присваивания =, имеют ассоциативность слева направо. Оператор присваивания (=) ассоциативен справа налево.

Обращайтесь к таблице приоритетов операторов, когда будете писать выражения с множеством операторов. Убедитесь, что операторы выполняются в правильном порядке. В случае каких-либо сомнений используйте круглые скобки для группировки подвыражений или для разбиения сложной инструкции на последовательность более простых инструкций. Помните, что некоторые операторы в языке C, такие как оператор присваивания (=), ассоциативны справа налево, а не слева направо.

Некоторые слова, использовавшиеся нами в программах на C в этой главе, в частности слово `int`, являются **ключевыми**, или зарезервированными, словами языка. В табл. 2.6 перечислены ключевые слова языка C. Эти слова имеют специальное значение для компилятора, поэтому их не следует использовать в качестве идентификаторов, таких как имена переменных.

Таблица 2.5 | Предшествование и ассоциативность операторов, обсуждавшихся выше

Операторы	Ассоциативность	Операторы	Ассоциативность
()	Слева направо	< <= > >=	Слева направо
* / %	Слева направо	== !=	Слева направо
+ -	Слева направо	=	Справа налево

Таблица 2.6 | Ключевые слова языка C

Ключевые слова			
`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`
Ключевые слова, добавленные стандартом C99			
`_Bool` `_Complex` `_Imaginary` `inline` `restrict`			
Ключевые слова, добавленные стандартом C11			
`_Alignas`	`_Alignof`	`_Atomic`	`_Generic`
`_Noreturn`	`_Static_assert`	`_Thread_local`	

2.6 Безопасное программирование на C

В предисловии к книге мы уже упоминали о стандарте «The CERT C Secure Coding Standard» и указывали, что будем следовать определенным нормам и правилам, помогающим избежать приемов программирования, открывающих систему для атак.

Не используйте printf с одним аргументом

Одним из таких правил является отказ от использования `printf` с единственным строковым аргументом. Если вам потребуется вывести строку, завершающуюся переводом строки, используйте функцию `puts`, которая выводит свой строковый аргумент и завершает вывод символом перевода строки. Например, строку 8 в примере 2.1

```
8    printf( "Welcome to C!\n" );
```

следовало бы записать так:

```
puts( "Welcome to C!" );
```

Мы убрали завершающий символ `\n` из сообщения, потому что функция `puts` добавляет его автоматически.

Если потребуется вывести строку, *без* перевода текстового курсора на новую строку, используйте `printf` с двумя аргументами – строкой формата `"%s"` и строкой для вывода на экран. Спецификатор преобразования `%s` указывает, что на экран должна быть выведена обычная текстовая строка. Например, инструкцию в строке 8 в примере 2.2

```
8    printf( "Welcome " );
```

следовало бы записать так:

```
printf( "%s", "Welcome " );
```

Хотя способы использования функции `printf` в этой главе *не* сопряжены с какими-либо опасностями, тем не менее описанные изменения соответствуют правилам, избавляющим от некоторых уязвимостей, с которыми мы познакомимся далее в этой книге. Начиная с этого момента, мы будем следовать данным правилам, и вы также должны следовать им в своем коде.

За дополнительной информацией об этой проблеме обращайтесь к правилу «CERT C Secure Coding rule FIO30-C»: `www.securecoding.cert.org/confluence/display/seccode/FIO30-C.+Exclude+user+input+from+format+strings`.

В разделе «Безопасное программирование на C» в главе 6 мы разъясним понятие ввода пользователя (user input) в соответствии с этим правилом в руководстве CERT.

scanf и printf, scanf_s и printf_s

В этой главе вы познакомились с функциями `scanf` и `printf`. Мы еще не раз будем возвращаться к ним в последующих разделах «Безопасное программирование на C». Мы также обсудим функции `scanf_s` и `printf_s`, введенные стандартом C11.

3

Управляющие инструкции: часть I

В этой главе вы познакомитесь:

- с инструкциями `if` и `if...else`, позволяющими выполнять или пропускать операции при определенных условиях;
- с инструкцией многократного повторения `while`, позволяющей повторно выполнять инструкции в программе;
- с инструкциями многократного повторения, управляемыми счетчиком или условным выражением;
- с приемами структурного программирования;
- с операторами инкремента, декремента и присваивания.

Краткое содержание

3.1 Введение

В следующих двух главах обсуждаются приемы, упрощающие разработку структурированных программ.

3.2 Управляющие структуры

Обычно инструкции в программе выполняются друг за другом, в порядке их следования. Это называется *последовательным выполнением*. Однако в языке C имеются различные инструкции, которые обсуждаются ниже, позволяющие указывать, какая инструкция должна быть выполнена следующей. Этот прием называется *передачей управления*.

Еще в 1960-х годах стало очевидно, что бесконтрольное использование инструкций передачи управления является корнем немалых проблем, испытываемых коллективами разработчиков. Виновницей была названа инструкция `goto`, позволявшая передавать управление практически в любую точку программы. В те годы новый термин «структурное программирование» стал почти синонимом «отказ от goto».

В своих исследованиях Бем (Bohm) и Якопини (Jacopini)[1] доказали, что любую программу можно написать *без* использования инструкции `goto`. Действительность требовала от программистов принять новый стиль «программирования без goto». Но по-настоящему новый стиль вошел в обиход только в 1970-х, когда к структурному программированию стали относиться серьезно. Результаты превзошли все ожидания: коллективы разработчиков все чаще заявляли об уменьшении сроков на разработку, все чаще системы стали поставляться вовремя, и все чаще стоимость разработки проектов стала укладываться в отведенный бюджет. Программы, произведенные с применением приемов структурного программирования, получались более понятными, более простыми в отладке и доработке, а количество первичных ошибок в них стало намного меньше.

[1] Bohm C., Jacopini G. Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules. – Communications of the ACM, Vol. 9, No. 5, May 1966, pp. 336–371.

В своей работе Бем (Bohm) и Якопини (Jacopini) показали, что любую программу можно написать в терминах всего трех управляющих структур, а именно *структуры последовательного выполнения, структуры выбора* и *структуры повторения*. Структура последовательного выполнения чрезвычайно проста – компьютер выполняет инструкции одну за другой, в порядке их следования в программе.

Блок-схемы

Блок-схема – это *графическое* представление алгоритма или его части. Блок-схемы рисуются с применением специальных элементов (блоков), таких как *прямоугольники*, *ромбы*, *прямоугольники с закруглениями* и *маленькие окружности*. Эти элементы соединяются между собой *стрелками*.

На рис. 3.1 представлен фрагмент блок-схемы, иллюстрирующий структуру последовательного выполнения. *Прямоугольные элементы*, также называемые *блоками операций*, или *блоками действий*, соответствуют каким-либо операциям или действиям, включая вычисления или ввод/вывод. Стрелки на рисунке указывают *порядок* выполнения операций – первая операция прибавляет значение `grade` к значению `total`, затем `1` прибавляется к значению `counter`. Структура последовательного выполнения может включать неограниченное количество операций. Как будет показано ниже, *везде, где можно вставить единственную операцию, можно вставить и множество операций, выполняемых последовательно*.

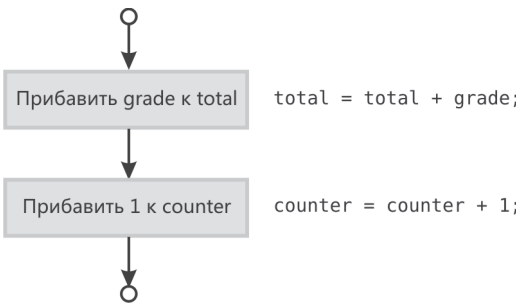


Рис. 3.1 | Блок-схема структуры последовательного выполнения

Когда блок-схема отражает *полный* алгоритм, она начинается *прямоугольником с закруглениями*, содержащим слово «Начало», а заканчивается *прямоугольником с закруглениями* со словом «Конец». Если блок-схема отражает только часть алгоритма, как на рис. 3.1, *прямоугольники с закруглениями* замещают окружностями, также называемыми *соединителями*.

Пожалуй, самый важный элемент блок-схемы – *ромб*, еще называемый *логическим блоком*, или *блоком условия*. Он указывает, что в данном месте алгоритма происходит принятие решения. Подробнее логический блок будет обсуждаться в следующем разделе.

Инструкции выбора в C

В языке C имеются три типа *структур выбора*, имеющих форму *инструкций*. Инструкция выбора `if` (раздел 3.3) *выбирает* (выполняет) действие, если условие *истинно*, или пропускает его, если условие *ложно*. Инструкция выбора `if...else` (раздел 3.4) выполняет одно действие, если условие *истинно*, и другое действие – если условие *ложно*. Инструкция выбора `switch` (обсуждается в главе 4) выполняет одно из *множества* разных действий в зависимости от значения управляющего выражения. Инструкцию `if` называют *инструкцией единственного выбора*, потому что она выбирает или игнорирует *единственное* действие. Инструкцию `if...else` называют *инструкцией двойного выбора*, потому что она выбирает между двумя разными действиями. Инструкцию `switch` называют *инструкцией множественного выбора*, потому что она делает выбор из множества действий.

Инструкции повторения в C

В языке C имеются три типа *структур повторения*, имеющих форму *инструкций*, а именно: инструкция `while`, инструкция `do...while` и инструкция `for`.

Это все, что есть в нашем распоряжении. В C имеются только *семь* управляющих структур: структура последовательного выполнения, три типа структур выбора и три типа структур повторения. Любая программа на языке C является комбинацией управляющих структур этих типов, соответствующих требованиям алгоритмов, реализуемых программой. На рис. 3.1 видно, что любая управляющая структура последовательного выполнения на блок-схеме представлена двумя окружностями, одна обозначает *точку входа*, и одна – *точку выхода*. Такие управляющие структуры с *единственной точкой входа* и *единственной точкой выхода* упрощают создание ясных и понятных программ. Сегменты блок-схемы могут соединяться друг с другом подключением точки выхода одной управляющей инструкции к точке входа другой. Мы называем это **связыванием управляющих структур**. Существует еще один способ связывания управляющих структур – *вложение друг в друга*. Итак, любую программу на языке C можно написать, используя всего лишь семь разных управляющих структур, связываемых между собой всего двумя способами. Это – основа простоты.

3.3 Инструкция выбора if

Инструкции выбора используются для выбора из имеющихся альтернатив. Например, допустим, что по 100-бальной системе удовлетворительной оценкой на экзамене считается оценка 60. Инструкция

```
if ( grade >= 60 ) {
    puts( "Passed" );
} // конец if
```

проверяет истинность условия `grade >= 60`. Если условие *истинно*, выводится строка «Passed» (Принято), и выполнение продолжается с инструкции, следующей далее. Если условие *ложно*, вывод строки пропускается, и выполнение продолжается сразу со следующей инструкции. Вторая строка в этой структуре выбора имеет отступ. Отступы в таких случаях необязательны, но они помогают видеть структуру программы. Компилятор C игнорирует *пробельные символы*, такие как пробелы, символы табуляции и перевода строки, используемые для оформления отступов и выделения логических блоков пустыми строками.

Блок-схема на рис. 3.2 иллюстрирует инструкцию единственного выбора `if`. Логический блок в форме ромба содержит выражение, играющее роль условия, которое может иметь либо истинное значение, либо ложное. Из логического блока выходят *две* стрелки. Одна соответствует *истинному* значению условного выражения в блоке, другая – *ложному*. Решения могут приниматься с применением операторов *отношений* или *проверки равенства*. В действительности решения могут приниматься на основе любых выражений: если выражение возвращает *нулевое значение*, оно интерпретируется как *ложное*, любое *ненулевое* значение интерпретируется как *истинное*.

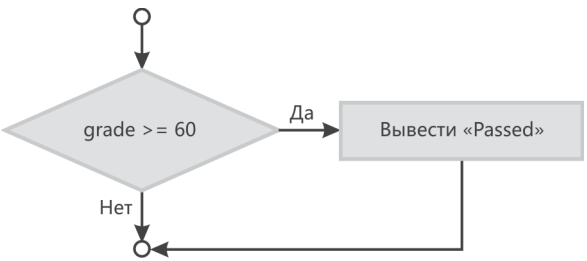


Рис. 3.2 | Блок-схема структуры единственного выбора

3.4 Инструкция выбора if...else

Инструкция выбора `if` выполняет указанное действие, только когда проверяемое условие истинно; в противном случае действие пропускается. Инструкция `if...else` позволяет указать два разных действия: одно выполняется, когда условие истинно, другое – когда условие ложно. Например, инструкция

```
if ( grade >= 60 ) {
    puts( "Passed" );
} // конец if
```

```
else {
    puts( "Failed" );
} // конец else
```

выведет "Passed", если учащийся получил оценку 60 или выше, и "Failed", если оценка ниже 60. В любом случае, после вывода той или иной строки выполнение будет продолжено с инструкции, следующей ниже. Тело ветви else также имеет отступ. Блок-схема на рис. 3.3 иллюстрирует, как протекает выполнение этой инструкции if...else.

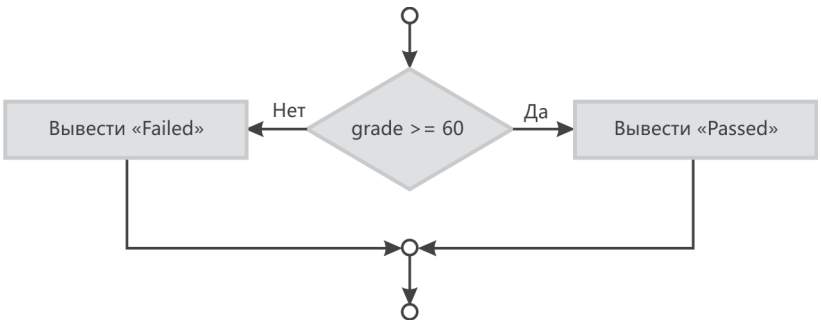


Рис. 3.3 | Блок-схема структуры двойного выбора

Условные операторы и условные выражения

В языке C имеется *условный оператор* (?:), который тесно связан с инструкцией if...else. Условный оператор – единственный *тернарный* (трехместный) оператор в языке C. Он принимает *три* операнда. Вместе с условным оператором они образуют **условное выражение**. Первый операнд – это *условие*. Второй операнд – значение всего условного выражения, когда условие *истинно*, и третий операнд – значение всего условного выражения, когда условие *ложно*. Например, инструкция puts

```
puts( grade >= 60 ? "Passed" : "Failed" );
```

получает в качестве аргумента условное выражение, которое возвращает строку "Passed", если условие grade >= 60 *истинно*, и строку "Failed", если условие *ложно*. Данная инструкция puts, по сути, выполняет то же самое, что и предыдущая инструкция if...else.

Второй и третий операнды в условном выражении могут быть также действиями. Например, условное выражение

```
grade >= 60 ? puts( "Passed" ) : puts( "Failed" );
```

читается так: «Если оценка grade больше или равна 60, выполнить puts("Passed"), в противном случае puts("Failed")». Эту инструкцию еще

можно сравнить с предыдущей инструкцией `if...else`. Далее в книге мы встретимся с ситуациями, когда условное выражение использовать можно, а инструкцию `if...else` – нет.

Вложенные инструкции if...else

Вложенные инструкции `if...else` позволяют проверить несколько условий путем вложения инструкций `if...else` друг в друга. Например, следующий код выведет `A`, если учащийся получил на экзамене оценку 90 или выше, `B` – если оценка выше или равна 80 (но ниже 90), `C` – если оценка выше или равна 70 (но ниже 80), `D` – если оценка выше или равна 60 (но ниже 70) и `F` – во всех остальных случаях.

```
if ( grade >= 90 ) {
    puts( "A" );
} // конец if
else {
    if ( grade >= 80 ) {
        puts("B");
    } // конец if
    else {
        if ( grade >= 70 ) {
            puts("C");
        } // конец if
        else {
            if ( grade >= 60 ) {
                puts( "D" );
            } // конец if
            else {
                puts( "F" );
            } // конец else
        } // конец else
    } // конец else
} // конец else
```

Если значение переменной `grade` больше или равно 90, все четыре условия будут *истинны*, но выполнена будет только первая инструкция `puts`. В этом случае ветвь `else` самой внешней инструкции `if...else` пропускается, и выполнение продолжается с первой инструкции, следующей за всем этим фрагментом кода.

Предыдущую инструкцию `if` можно записать иначе:

```
if ( grade >= 90 ) {
    puts( "A" );
} // конец if
else if ( grade >= 80 ) {
    puts( "B" );
} // конец else if
else if ( grade >= 70 ) {
    puts( "C" );
} // конец else if
else if ( grade >= 60 ) {
    puts( "D" );
} // конец else if
else {
    puts( "F" );
} // конец else
```

С точки зрения компилятора C обе формы эквивалентны. Однако последняя пользуется большей популярностью, потому что она избавляет от необходимости увеличивать отступы все больше и больше. Из-за больших отступов в строке может остаться слишком мало места, что будет вызывать необходимость разбивать инструкции на несколько строк и ухудшать читаемость программы.

Инструкция выбора `if` позволяет указать в теле лишь *одну* инструкцию, поэтому, если у вас тело инструкции `if` состоит из единственной инструкции, фигурные скобки можно *опустить*. Чтобы включить в тело `if` несколько инструкций, их следует заключить в фигурные скобки (`{` и `}`). Множество инструкций, заключенных в пару фигурных скобок, называется **составной инструкцией**, или **блоком**.

Составную инструкцию можно использовать везде, где можно использовать единственную инструкцию.

Следующий фрагмент включает составную инструкцию в ветви `else` инструкции `if...else`.

```
if ( grade >= 60 ) {
    puts( "Passed." );
} // конец if
else {
    puts( "Failed." );
    puts( "You must take this course again." );
} // конец else
```

В данном случае, если оценка меньше 60, программа выполнит обе инструкции `puts` в теле ветви `else` и выведет

```
Failed.
You must take this course again.
```

Фигурные скобки, окружающие две инструкции в ветви `else`, играют важную роль. Без них инструкция

```
puts( "You must take this course again." );
```

оказалась бы *за пределами* тела ветви `else` и выполнялась бы *всегда*, независимо от оценки за экзамен.

На место единственной инструкции не только можно подставить составную инструкцию, можно также вообще убрать эту инструкцию, то есть вставить **пустую инструкцию**. Роль пустой инструкции играет точка с запятой (`;`).

Размещение точки с запятой после условия в инструкции `if`, как в инструкции `if ( grade >= 60 );`, приведет к логической ошибке в инструкции `if` единственного выбора и к синтаксической ошибке в инструкции `if` двойного выбора.

3.5 Инструкция повторения while

Инструкция повторения `while` (также называется **инструкцией цикла**) позволяет указать действие, которое должно *повторяться*, пока условие продолжает оставаться *истинным*. В конечном счете условие должно стать *ложным*. В этот момент повторения прекратятся, и выполнение продолжится с первой инструкции, следующей за инструкцией повторения.

В качестве иллюстрации применения инструкции **`while`** рассмотрим фрагмент программы, отыскивающий первую степень тройки больше 100. Допустим, что целочисленная переменная инициализирована значением 3. Когда следующая инструкция повторения `while` завершит выполнение, переменная `product` будет содержать требуемый ответ:

```
product = 3;

while ( product <= 100 ) {
    product = 3 * product;
} // конец while
```

Тело инструкции повторения while

Инструкции, находящиеся внутри инструкции повторения `while`, образуют ее тело. Телом инструкции `while` может быть единственная инструкция или составная инструкция, заключенная в фигурные скобки (`{` и `}`).

Если в теле инструкции `while` не выполнять действий, которые в конечном итоге обеспечат ложность условия, цикл может стать бесконечным.

Блок-схема инструкций повторения

Блок-схема на рис. 3.4 иллюстрирует, как протекает выполнение инструкции повторения `while`. Здесь отчетливо видны повторения. Стрелка выходит из прямоугольника действия в цикле и опять ведет в точку принятия решения, где выполняется проверка условия перед началом нового цикла.

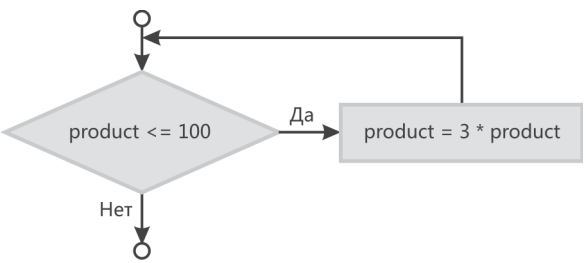


Рис. 3.4 | Блок-схема инструкции повторения while

Когда условие оказывается ложным, инструкция `while` завершается, и управление передается следующей инструкции в программе.

На входе в инструкцию `while` переменная `product` имеет значение 3. В каждом повторении переменная `product` умножается на 3, поочередно принимая значения 9, 27 и 81. Когда `product` получает значение 243, условие в инструкции `while`, `product <= 100`, становится ложным. В результате повторения прекращаются, и в переменной `product` остается последнее значение 243, а выполнение программы продолжается с инструкции, следующей за `while`.

3.6 Определение средней оценки с помощью инструкции повторения, управляемой счетчиком

Далее мы попробуем решить задачу поиска средней оценки по классу. Вот как описывается эта задача:

> *В классе с десятью учащимися был проведен опрос. Оценки (целые числа в диапазоне от 0 до 100), полученные учащимися, доступны вам. Определите среднюю оценку по классу.*

Средняя оценка по классу вычисляется как сумма всех оценок, деленная на количество учащихся. Программа, решающая эту задачу, вводит каждую оценку, вычисляет среднее значение и выводит его (пример 3.1).

Для ввода оценок, по одной за раз, мы использовали *инструкцию повторения, управляемую счетчиком*. В данном решении используется счетчик, определяющий количество повторений: повторения прекращаются, когда значение счетчика станет больше 10.

Пример 3.1 | Задача определения средней оценки с помощью инструкции повторения, управляемой счетчиком

```
1 // Пример 3.1: fig03_05.c
2 // Вычисление средней оценки инструкцией повторения, управляемой счетчиком
3 #include <stdio.h>
4
5 // выполнение программы начинается с функции main
6 int main( void )
7 {
8    unsigned int counter; // количество оценок
9    int grade;            // значение оценки
10   int total;            // сумма оценок
11   int average;          // средняя оценка
12
13   // фаза инициализации
14   total = 0;            // инициализация суммы
15   counter = 1;          // initialize loop counter
```

```
16
17    // фаза обработки
18    while( counter <= 10 ){             // повторить 10 раз
19       printf( "%s", "Enter grade: " ); // приглашение к вводу
20       scanf( "%d", &grade );           // прочитать оценку
21       total = total + grade;           // сложить grade и total
22       counter = counter + 1;           // увеличить счетчик
23    } // конец while
24
25    // фаза завершения
26    average = total / 10;               // целочисленное деление
27
28    printf( "Class average is %d\n", average ); // вывести результат
29 } // конец функции main
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Мы используем сумму и счетчик. Так как переменная счетчика `counter` хранит только числа от 1 до 10 (положительные), мы объявили ее как переменную типа `unsigned int`, способную хранить лишь неотрицательные значения (то есть 0 и выше).

В результате вычислений в программе средняя оценка получилась равной 81. В действительности сумма всех оценок равна 817. То есть при делении на 10 должно было получиться число 81.7 – *число с точкой*. Как обрабатывать такие *дробные числа*, мы расскажем в следующем разделе.

3.7 Определение средней оценки с помощью инструкции повторения, управляемой сигнальным значением

Давайте обобщим задачу вычисления средней оценки. Переформулируем ее:

> *Разработать программу вычисления средней оценки по классу для произвольного количества учащихся.*

В первом примере количество учеников в классе было известно заранее. В этом примере количество оценок неизвестно. Программа должна обрабатывать *любое* количество оценок.

Одним из решений в подобных ситуациях является использование специального, *сторожевого* значения (также называемого *сигнальным*, или *флаговым*), отмечающего «конец ввода данных». Пользователь продолжает вводить оценки, пока они не закончатся, а затем вводит сигнальное значение, чтобы показать, что «ввод оценок завершился».

Очевидно, что сигнальное значение должно выбираться таким, чтобы его нельзя было спутать с допустимым значением. Поскольку обычно роль оценок играют *неотрицательные* целые числа, значение –1 будет отличным сигнальным значением в данной задаче. То есть программа вычисления средней оценки может получить последовательность чисел 95, 96, 75, 74, 89 и –1, и в ответ на нее она должна вычислить и вывести среднее значение от оценок 95, 96, 75, 74 и 89 (сигнальное значение –1 *не* должно участвовать в вычислениях).

Исходный текст программы и результаты ее использования приводятся в примере 3.2. Несмотря на то что оценки являются целыми числами, при вычислении среднего часто могут получаться *числа с десятичной точкой*. Тип `int` не может представлять подобные числа. Поэтому в программе был использован новый тип данных **`float`** – тип чисел с плавающей точкой.

Пример 3.2 | Задача определения средней оценки с помощью инструкции повторения, управляемой сигнальным значением

```
// Пример 3.2: fig03_06.c
// Инструкция повторения, управляемая сигнальным значением.
#include <stdio.h>

// выполнение программы начинается с функции main
int main( void )
{
   unsigned int counter; // количество оценок
   int grade;            // значение оценки
   int total;            // сумма оценок

   float average;        // средняя оценка - число с плавающей точкой

   // фаза инициализации
   total = 0;            // инициализация суммы
   counter = 0;          // инициализация счетчика

   // фаза обработки
   // получить первую оценку
   printf( "%s", "Enter grade, -1 to end: " ); // приглашение к вводу
   scanf( "%d", &grade );                      // прочитать оценку

   // цикл продолжается, пока не введено сигнальное значение
   while( grade != -1 ){
      total = total + grade; // сложить grade и total
      counter = counter + 1; // увеличить счетчик

      // получить следующую оценку
```

```
29       printf( "%s", "Enter grade, -1 to end: " ); // приглашение к вводу
30       scanf( "%d", &grade );                      // прочитать оценку
31    } // конец while
32
33    // фаза завершения
34    // если пользователь ввел хотя бы одну оценку
35    if( counter != 0 ){
36
37       // вычислить среднее для введенных оценок
38       average = ( float ) total / counter; // чтобы избежать усечения
39
40       // вывести среднее с точностью до второго знака после запятой
41       printf( "Class average is \n", average );
42    } // конец if
43    else { // если ни одной оценки не введено, вывести сообщение
44       puts( "No grades were entered" );
45    } // конец else
46 } // конец функции main
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

```
Enter grade, -1 to end: -1
No grades were entered
```

Обратите внимание на составную инструкцию в цикле `while` (строка 24) в примере 3.2. Фигурные скобки здесь *совершенно необходимы*, чтобы гарантировать выполнение в цикле всех четырех инструкций. Без фигурных скобок последние три инструкции *выпали* бы из цикла, и тогда компьютер интерпретировал бы наш код, как показано ниже:

```
24    while( grade != -1 )
25       total = total + grade; // сложить grade и total
26    counter = counter + 1; // увеличить счетчик
27
28    // получить следующую оценку
29    printf( "%s", "Enter grade, -1 to end: " ); // приглашение к вводу
30    scanf( "%d", &grade );                      // прочитать оценку
```

В этом случае программа будет попадать в *бесконечный цикл*, если только в качестве самой первой оценки не передать ей сигнальное значение `-1`.

В цикле, управляемом сигнальным значением, приглашение к вводу должно содержать явное напоминание значения, используемого в качестве сигнального.

Явное и неявное преобразования типов

В результате вычисления среднего не всегда получается целое число. Часто получается вещественное значение, такое как 7.2 или –93.5, которое может быть представлено типом данных `float`. Переменная `average` объявлена с типом `float` (строка 12) и способна хранить дробные результаты вычислений. Однако результатом операции `total / counter` является целое число, потому что *обе* переменные, `total` и `counter`, являются целочисленными. При делении целых чисел выполняется *целочисленное деление*, возвращающее целое число, от которого *отбрасывается* дробная часть. Поскольку вычисление выполняется до того, как результат будет присвоен переменной `average`, дробная часть теряется безвозвратно. Чтобы получить вещественное число уже на этапе вычисления, необходимо создать *временные* вещественные значения. Для этой цели в языке C имеется **унарный оператор приведения типа**. Строка 38

```
38       average = ( float ) total / counter; // чтобы избежать усечения
```

включает оператор `(float)`, создающий *временную* вещественную копию своего операнда, `total`. Значение в переменной `total` по-прежнему остается целочисленным. Оператор приведения типа выполняет **явное преобразование**. Теперь в вычислениях участвуют вещественное число (временная версия типа `float` переменной `total`) и беззнаковое целое из переменной `counter`. В языке C арифметические выражения вычисляются только с операндами *идентичных* типов. Чтобы гарантировать *идентичность*, компилятор выполнит над выбранным операндом операцию, называемую **неявным приведением типа**. Например, в выражении, содержащем операнды типов `unsigned int` и `float`, копия операнда типа `unsigned int` будет приведена к типу `float`. В нашем примере будет создана копия значения переменной `counter` и приведена к типу `float`, затем будет выполнена операция деления, и ее результат (вещественное число) будет присвоен переменной `average`. В языке C имеется отдельный свод правил преобразования типов операндов. Мы рассмотрим их в главе 5.

Операторы приведения доступны для большинства типов данных – они образуются заключением имени типа в круглые скобки. Все операторы приведения типа являются *унарными*, то есть принимают только один операнд. В главе 2 мы познакомились с двухместными арифметическими операторами. В языке C имеются также унарные версии операторов «плюс» (`+`) и «минус» (`–`), поэтому вы можете записывать выражения, такие как `+7` или `–5`. Операторы приведения типов ассоциативны справа налево и имеют тот же приоритет, что и унарные операторы, такие как `+` и `–`. Их приоритет на одну ступень выше приоритета *мультипликативных операторов* `*`, `/` и `%`.

Форматирование вещественных чисел

В примере 3.1, в вызове функции `printf`, использован спецификатор преобразования `%.2f` (строка 41) для вывода значения `average`. Символ `f` ука-

зывает, что будет выводиться вещественное (floating-point) значение. Часть `.2` определяет *точность* отображения значения – 2 знака после десятичной точки. Если бы использовался спецификатор `%f` (без указания точности), использовалась бы *точность по умолчанию* 6, как если бы был указан спецификатор `%.6f`. При выводе вещественных значений с определенной точностью выводимое значение *округляется* до указанного знака после десятичной точки. Значение, хранящееся в памяти, при этом не изменяется. Например, следующие инструкции выведут значения 3.45 и 3.4.

```
printf( "%.2f\n", 3.446 ); // выведет 3.45
printf( "%.1f\n", 3.446 ); // выведет 3.4
```

Использование спецификатора преобразования с точностью в инструкции `scanf` *является ошибкой. Точность может указываться только для вывода, в функции* `printf`.

Примечания относительно вещественных чисел

Несмотря на то что вещественные числа не всегда обеспечивают «точность на все 100%», они имеют массу применений. Например, когда мы говорим о «нормальной» температуре тела 36,6 °C, мы не имеем в виду точность до большого количества знаков после запятой. Когда мы рассматриваем показания термометра и читаем их как 36,6 °C, фактическая температура может быть равной 36,5999473210643 °C. Но часто нам достаточно назвать более простое число 36,6. Подробнее об этой проблеме мы поговорим позже.

Еще одна проблема вещественных чисел кроется в операции деления. Когда мы делим 10 на 3, в результате получается число 3,3333333... с бесконечной последовательностью троек. Однако для хранения вещественных чисел компьютер выделяет *ограниченный* объем памяти, поэтому в памяти сохраняется *приближенное* значение фактического результата.

Если при использовании вещественных чисел исходить из предположения их абсолютной точности, это может приводить к ошибочным результатам. В большинстве случаев компьютеры хранят лишь приближенные значения.

Не проверяйте вещественные числа на равенство.

3.8 Вложенные управляющие инструкции

Мы узнали, что управляющие инструкции могут *объединяться* друг с другом (последовательно). В следующем примере мы познакомимся еще с одним структурированным способом связывания управляющих инструкций – посредством *вложения* одной управляющей инструкции *в* другую.

Взгляните на следующее описание задачи:

> *Колледж предлагает курс подготовки учащихся к государственному экзамену по специальности «брокер». В прошлом году 10 учащихся приступили к обучению на данном курсе и подошли к экзаменационным испытаниям. Вполне естественно, что руководство колледжа хотело бы знать, насколько успешно студенты сдают экзамены. Вам было предложено написать программу, суммирующую результаты. В вашем распоряжении имеется список из 10 учащихся. Напротив каждой фамилии стоит 1, если учащийся сдал экзамен, и 2, если не сдал.*
>
> *Ваша программа должна проанализировать результаты экзаменов, как описано ниже:*
>
> 1. *Для ввода результата каждого учащегося (то есть 1 или 2) программа должна отобразить строку приглашения к вводу «Enter result» (Введите результат) и дождаться ввода пользователя.*
> 2. *Подсчитать количество результатов каждого типа.*
> 3. *Вывести результаты, сообщив количество учащихся, сдавших и не сдавших экзамен.*
> 4. *Если количество сдавших окажется больше восьми, вывести сообщение «Bonus to instructor!» (Выписать премию преподавателю!).*

После внимательного прочтения требований мы пришли к следующим выводам:

1. Программа должна обработать 10 результатов экзаменов, поэтому будет использоваться цикл, управляемый счетчиком.
2. Все результаты являются числами – либо 1, либо 2. Для каждого введенного результата программа должна определить, является ли число 1 или 2. В нашем алгоритме мы будем сравнивать введенный результат с числом 1. Если введенное число не равно 1, предполагается 2. В качестве самостоятельного упражнения попробуйте изменить программу так, чтобы она проверяла результат на равенство 1 или 2 и выводила бы сообщение об ошибке, если пользователь ввел какое-то другое число.
3. В программе будут использоваться два счетчика – один для подсчета количества учащихся, сдавших экзамен, и один для подсчета количества учащихся, не сдавших экзамена.
4. После обработки результатов программа должна проверить, не превысило ли количество учащихся, сдавших экзамен, число 8.

Исходный текст программы и два листинга сеансов работы с ней представлены в примере 3.3. Мы воспользовались особенностью языка C, позволяющей совмещать инициализацию с объявлением (строки 9–11). Такого рода инициализация выполняется на этапе компиляции. Обратите также внимание, что для вывода беззнакового целого (`unsigned int`) используется спецификатор преобразования `%u` (строки 33–34).

Пример 3.3 | Анализ результатов экзаменов

```
 1 // Пример 3.3: fig03_07.c
 2 // Анализ результатов экзаменов.
 3 #include <stdio.h>
 4
 5 // выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    // инициализация переменных в объявлениях
 9    unsigned int passes   = 0; // количество сдавших экзамен
10    unsigned int failures = 0; // количество не сдавших экзамена
11    unsigned int student  = 1; // количество студентов
12    int result;                // результат одного учащегося
13
14    // обработать 10 результатов с помощью цикла, управляемого счетчиком
15    while ( student <= 10 ) {
16
17       // вывести приглашение к вводу и ввести результат
18       printf( "%s", "Enter result ( 1=pass,2=fail ): " );
19       scanf( "%d", &result );
20
21       // если введена 1, увеличить количество сдавших
22       if ( result == 1 ) {
23          passes = passes + 1;
24       } // конец if
25       else { // иначе увеличить количество не сдавших
26          failures = failures + 1;
27       } // конец else
28
29       student = student + 1; // увеличить счетчик учащихся
30    } // конец while
31
32    // завершающая фаза; вывести количество сдавших и не сдавших
33    printf( "Passed %u\n", passes );
34    printf( "Failed %u\n", failures );
35
36    // если сдавших больше восьми, вывести "Bonus to instructor!"
37    if ( passes > 8 ) {
38       puts( "Bonus to instructor!" );
39    } // конец if
40 } // конец функции main
```

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4
```

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Passed 9
Failed 1
Bonus to instructor!
```

3.9 Операторы присваивания

В языке C имеется несколько операторов присваивания, совмещенных с операциями. Например, инструкцию

```
c = c + 3;
```

можно записать в более краткой форме с помощью оператора `+=`:

```
c += 3;
```

Оператор `+=` прибавляет значение выражения справа к значению переменной слева и сохраняет результат в этой переменной. Любая инструкция вида

```
variable = variable operator expression;
```

где роль *`operator`* может играть любой из двухместных операторов `+`, `-`, `*`, `/` и `%` (а также другие, которые мы перечислим в главе 10), может быть записана в виде

```
variable operator= expression;
```

То есть инструкция `c += 3` прибавит `3` к `c`. В табл. 3.1 перечислены комбинированные операторы присваивания, примеры использования этих операторов в выражениях и краткое описание.

Таблица 3.1 | Комбинированные арифметические операторы присваивания

Оператор	Пример выражения	Описание	Сохранит
Предполагается: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10 в c`
`-=`	`d -= 4`	`d = d -4`	`1 в d`
`*=`	`e *= 5`	`e = e * 5`	`20 в e`
`/=`	`f /= 3`	`f = f / 3`	`2 в f`
`%=`	`g %= 9`	`g = g % 9`	`3 в g`

3.10 Операторы инкремента и декремента

В языке C также имеются **унарный оператор инкремента**, ++, и **унарный оператор декремента**, --, описание которых приводится в табл. 3.2. Если необходимо увеличить значение переменной c на единицу, вместо выражения c = c + 1 или c += 1 можно использовать оператор ++. Оператор инкремента или декремента, стоящий *перед* переменной, называют **префиксным** оператором инкремента или декремента соответственно. Оператор инкремента или декремента, стоящий *после* переменной, называют **постфиксным** оператором инкремента или декремента соответственно. Префиксный оператор сначала увеличивает (уменьшает) значение переменной на 1 и возвращает новое значение. Постфиксный оператор также увеличивает (уменьшает) значение переменной на 1, но в качестве результата возвращает прежнее ее значение.

Таблица 3.2 | Операторы инкремента и декремента

Оператор	Пример выражения	Описание
++	++a	Увеличит переменную a на 1 и вернет новое значение переменной
++	a++	Увеличит переменную a на 1 и вернет старое значение переменной
--	--b	Уменьшит переменную b на 1 и вернет новое значение переменной
--	b--	Уменьшит переменную b на 1 и вернет старое значение переменной

Отличия префиксного и постфиксного операторов инкремента

В примере 3.4 демонстрируются отличия префиксной и постфиксной версий оператора инкремента ++. При использовании постфиксного инкремента значение переменной увеличивается *после* ее использования в инструкции printf. Соответственно, при использовании префиксного инкремента значение переменной увеличивается *до* использования в инструкции printf.

Пример 3.4 | Префиксный и постфиксный операторы инкремента

```
1 // Пример 3.4: fig03_10.c
2 // Префиксный и постфиксный операторы инкремента.
3 #include <stdio.h>
4
5 // выполнение программы начинается с функции main
6 int main( void )
7 {
8    int c;                // определение переменной
```

```
 9
10    // демонстрация префиксного инкремента
11    c = 5;                  // присвоить значение 5 переменной c
12    printf( "%d\n", c );    // выведет 5
13    printf( "%d\n", c++ ); // выведет 5, затем увеличит переменную c
14    printf( "%d\n\n", c ); // выведет 6
15
16    // демонстрация постфиксного инкремента
17    c = 5;                  // присвоить значение 5 переменной c
18    printf( "%d\n", c );    // выведет 5
19    printf( "%d\n", ++c ); // увеличит переменную c и выведет 6
20    printf( "%d\n\n", c ); // выведет 6
21 } // конец функции main
```

```
5
5
6

5
6
6
```

Программа выводит значение переменной `c` до и после применения оператора `++`. Оператор декремента (`--`) действует аналогично.

Унарные операторы должны размещаться вплотную к своим операндам, без пробелов между ними.

Три инструкции присваивания из примера 3.3:

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

Можно было бы записать более кратко, с помощью *комбинированных операторов присваивания*:

```
passes += 1;
failures += 1;
student += 1;
```

с помощью префиксного оператора инкремента:

```
++passes;
++failures;
++student;
```

или с помощью постфиксного оператора инкремента:

```
passes++;
failures++;
student++;
```

Важно отметить, что если оператор инкремента или декремента используется как *самостоятельная* инструкция, префиксная или постфиксная форма

дает *одинаковый* эффект. Но когда эти операторы применяются в контексте более крупного выражения, префиксная или постфиксная форма производит *разный* эффект. Из выражений, которые мы видели до сих пор, явственно следует, что операторы инкремента и декремента могут применяться только к простым переменным.

Попытка применить оператор инкремента или декремента к выражению, отличному от простой переменой, вызывает синтаксическую ошибку, например `++(x + 1)`.

Обычно в языке C не определяется порядок обработки операндов операторами (хотя в главе 4 будет показано несколько исключений из этого правила). Поэтому операторы инкремента и декремента следует использовать только в отдельных инструкциях увеличения или уменьшения переменной.

В табл. 3.3 перечислены операторы, их приоритеты и ассоциативность, с которыми мы познакомились к настоящему моменту. Операторы перечислены сверху вниз, в порядке уменьшения приоритетов. Во втором столбце указывается ассоциативность операторов на каждом уровне приоритета. Обратите внимание, что унарные операторы инкремента (`++`), декремента (`--`), плюс (`+`), минус (–), приведения типа, операторы присваивания (`=`, `+=`, `-=`, `*=`, `/=` и `%=`) и условный оператор (`?:`) ассоциативны справа налево. В третьем столбце указаны названия различных групп операторов. Все остальные операторы в табл. 3.3 ассоциативны слева направо.

Таблица 3.3 | Приоритеты и ассоциативность операторов, представленных к настоящему моменту

Операторы	Ассоциативность	Тип
++ (постфиксный) -- (постфиксный)	Справа налево	Постфиксный
+ - (тип) ++ (префиксный) -- (префиксный)	Справа налево	Унарный
* / %	Слева направо	Мультипликативный
+ -	Слева направо	Аддитивный
< <= > >=	Слева направо	Проверка отношения
== !=	Слева направо	Проверка на равенство
?:	Справа налево	Условный
= += -= *= /= %=	Справа налево	Присваивание

3.11 Безопасное программирование на C

Арифметическое переполнение

В примере 2.4 представлена программа, вычисляющая сумму двух целочисленных значений (строка 18) с инструкцией

```
18    sum = integer1 + integer2;          // вычислить и сохранить сумму
```

Даже в такой простой инструкции заключена потенциальная проблема – при сложении целых чисел результат может оказаться *больше*, чем способна вместить переменная типа `int`. Эта проблема известна как **арифметическое переполнение** и может стать причиной неожиданного поведения программы вплоть до создания бреши в системе безопасности.

Максимальное и минимальное значения, которые могут хранить переменные типа `int`, определены константами `INT_MAX` и `INT_MIN` соответственно, объявленными в заголовочном файле `<limits.h>`. Аналогичные константы существуют и для других целочисленных типов, с ними мы познакомимся в главе 4. Увидеть эти значения для своей платформы можно в заголовочном файле `<limits.h>`, открыв его в текстовом редакторе.

Считается хорошей практикой перед операциями, подобными той, что выполняется в примере 2.4 в строке 18, проверять возможность переполнения. Код, выполняющий такую проверку, показан на веб-сайте центра CERT `www.securecoding.cert.org` – просто поищите по названию «INT32-C». Этот код использует операторы `&&` (логическое И) и `||` (логическое ИЛИ), с которыми мы познакомимся в главе 4. В промышленном коде подобные проверки желательно выполнять перед *любыми* вычислениями. В следующих главах будут представлены другие приемы обработки подобных ошибок.

Деление на ноль

В примере вычисления средней оценки по классу существует вероятность, что делитель получит нулевое значение. В промышленных приложениях подобное допускать никак нельзя.

Перед выполнением деления на выражение, которое может вернуть нулевое значение, следует явно проверить эту ситуацию и обработать ее (например, вывести сообщение об ошибке), чтобы не допустить аварийного завершения программы.

Беззнаковые целые

В примере 3.1 в строке 8 переменная `counter` объявлена как `unsigned int`, потому что она может принимать *только неотрицательные значения*. Вообще, подобные счетчики, которые не должны принимать отрицательные значения, всегда следует объявлять с ключевым словом `unsigned` перед именем целочисленного типа. Переменные беззнаковых типов могут содержать значения от 0 до положительного значения, почти в два раза превышающего максимальное значение соответствующего целочисленного типа со знаком. Максимальное целое значение без знака (для типа `unsigned int`) определено в виде константы `UINT_MAX` в файле `<limits.h>`.

Целочисленные переменные `grade`, `total` и `average` в примере 3.1 также можно было бы объявить беззнаковыми. Оценки обычно находятся в диапазоне от 0 до 100, поэтому их сумма (`total`) и среднее (`average`) также будут больше или равны 0. Мы объявили их с типом `int` только потому, что не

можем контролировать ввод пользователя – пользователь может вводить и отрицательные значения. Хуже того, пользователь может даже вводить значения, вообще не являющиеся числами. (Как обрабатывать подобные ситуации, мы покажем далее в этой книге.)

Иногда для завершения цикла, управляемого сигнальным значением, используются заведомо недопустимые значения. Например, программа вычисления средней оценки по классу (пример 3.2) завершает цикл, когда пользователь введет сигнальное значение `-1` (которое не может быть оценкой), поэтому было бы неправильным объявить переменную `grade` с типом `unsigned int`. Как будет показано далее, признак конца файла (`EOF`) – мы познакомимся с ним в следующей главе, – который часто используется для завершения циклов, управляемых сигнальным значением, также является отрицательным числом. За дополнительной информацией обращайтесь к главе 5 «Integer Security» в книге Роберта Сикорда (Robert Seacord) «Secure Coding in C and C++».

scanf_s и printf_s

В приложении Annex K к стандарту C11 вводятся более защищенные версии функций `printf` и `scanf` с именами `printf_s` и `scanf_s`. Приложение Annex K является необязательным для реализации, поэтому не все разработчики компиляторов языка C будут его поддерживать.

В корпорации Microsoft были реализованы собственные версии `printf_s` и `scanf_s`, еще до публикации стандарта C11, и с того момента компилятор стал выводить предупреждения, встречая вызовы `scanf` в тексте программ. Текст предупреждения сообщает, что функция `scanf` относится к разряду *нежелательных* (deprecated), что ее не следует больше использовать и желательно заменить ее функцией `scanf_s`.

Многие организации вводят собственные стандарты кодирования операций, которые требуют, чтобы код компилировался *без вывода предупреждений компилятором*. В Visual C++ существует два способа избавиться от предупреждений, связанных с использованием функции `scanf`, – заменить ее функцией `scanf_s` и запретить вывод предупреждений. В инструкциях ввода, применявшихся до сих пор, пользователи Visual C++ могут просто заменить `scanf` на `scanf_s`. Запретить вывод предупреждений компилятором Visual C++ можно следующим образом:

1. Нажмите комбинацию клавиш *Alt+F7*, чтобы вывести диалог **Property Pages** (**Параметры проекта**) с настройками проекта.
2. В левой панели распахните пункт **Configuration Properties** > **C/C++** (**Настройки** > **C/C++**) и выберите пункт **Preprocessor** (**Препроцессор**).
3. В правой панели, в конце значения **Preprocessor Definitions** (**Определения препроцессора**), вставьте строку:

```
;_CRT_SECURE_NO_WARNINGS
```

4. Щелкните на кнопке **OK**, чтобы сохранить изменения.

После этого предупреждения, касающиеся использования функции `scanf` (и любых других, которые в Microsoft объявлены нежелательными по аналогичным причинам), перестанут появляться. При разработке промышленных приложений не рекомендуется запрещать вывод предупреждений. Мы еще будем говорить о функциях `scanf_s` и `printf_s` в последующих разделах «Безопасное программирование на C».

4

Управляющие инструкции: часть II

В этой главе вы познакомитесь:

- с основами использования инструкций повторения, управляемых счетчиками;
- с инструкциями повторения `for` и `do...while`;
- с инструкцией множественного выбора `switch`;
- с инструкциями `break` и `continue`, осуществляющими передачу управления;
- с использованием логических операторов для составления сложных условных выражений в управляющих инструкциях;
- с приемами, позволяющими избежать путаницы между операторами сравнения и присваивания.

Краткое содержание

4.1 Введение

В этой главе более подробно будут рассматриваться уже знакомые вам, а также дополнительные инструкции управления повторениями, а именно `for` и `do...while`. Помимо этого, будет представлена инструкция множественного выбора `switch`. Затем мы обсудим инструкцию `break`, обеспечивающую немедленный выход из инструкций управления, и инструкцию `continue`, позволяющую пропустить остаток тела инструкции повторения и перейти к следующей итерации. В этой главе также обсуждаются логические операторы, используемые для объединения условий, и повторно перечисляются принципы структурного программирования, упоминаемые в главах 3 и 4.

4.2 Основы повторения

Большинство программ используют инструкции повторения, или циклы. Цикл – это группа инструкций, которые компьютер выполняет многократно, пока некоторое **условие продолжения цикла** остается истинным. В предыдущей главе мы обсудили две разновидности повторений:

- управляемые счетчиками;
- управляемые сигнальным значением.

Повторение, управляемое счетчиком, иногда называют *повторением с ограниченным количеством итераций* (definite repetition), потому что заранее точно известно, сколько раз выполнится цикл. Повторение, управляемое сигнальным значением, иногда называют *повторением с неограниченным количеством итераций* (indefinite repetition), потому что это количество заранее неизвестно.

В инструкциях повторения со счетчиком для подсчета итераций используется **управляющая переменная** (control variable). Значение управляющей переменной увеличивается (обычно на 1) каждый раз, когда выполняется повторяемая группа инструкций. Когда значение управляющей переменной достигнет указанного числа, цикл завершается и выполнение продолжается с первой инструкции, следующей за инструкцией повторения.

Повторение с сигнальным значением используется, когда:

1) точное количество повторений неизвестно заранее;
2) цикл включает инструкции, извлекающие данные в каждой итерации.

Сигнальное значение указывает на «конец данных» и следует после всех обычных элементов данных, получаемых программой. Сигнальные значения должны отличаться от обычных данных.

4.3 Повторение со счетчиком

Для организации повторений со счетчиком необходимо определить:

1) **имя** управляющей переменной (или счетчика цикла);
2) **начальное значение** управляющей переменной;
3) операцию увеличения (или уменьшения) значения управляющей переменной в каждом цикле;
4) условие, проверяющее конечное значение управляющей переменной (то есть возможность продолжения цикла).

Рассмотрим программу в примере 4.1, которая выводит числа от 1 до 10. Инструкция

```
unsigned int counter = 1; // инициализация
```

объявляет управляющую переменную (`counter`) как целочисленную, резервирует место в памяти для ее хранения и устанавливает начальное значение, равное 1.

Пример 4.1 | Повторение, управляемое счетчиком

```
 1 // Пример 4.1: fig04_01.c
 2 // Повторение, управляемое счетчиком.
 3 #include <stdio.h>
 4
 5 // выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    unsigned int counter = 1; // инициализация
 9
10    while( counter <= 10 ){        // условие повторения
11       printf ( "%u\n", counter ); // вывести счетчик
12       ++counter;                  // увеличить
13    } // конец while
14 } // конец функции main
```

```
1
2
3
4
```

```
5
6
7
8
9
10
```

Объявить и инициализировать счетчик `counter` можно было бы немного иначе:

```
unsigned int counter;
counter = 1;
```

Определение переменной *не* генерирует выполняемый код, *в отличие от* инструкции присваивания. Вы можете использовать любой метод установки значений переменных, какой вам понравится.

Инструкция

```
++counter;                  // увеличить
```

увеличивает счетчик цикла на 1 в каждой итерации. Условие продолжения цикла в инструкции `while` проверяет, не превысило ли значение управляющей переменной числа `10` (последнее значение, для которого условное выражение остается истинным). Тело этого цикла выполняется, даже когда значение управляющей переменной равно 10. А когда оно *превысит* 10 (то есть когда переменная `counter` получит значение `11`), цикл завершится.

Программу в примере 4.1 можно было бы сократить, инициализировав переменную `counter` значением `0` и изменив инструкцию `while`, как показано ниже:

```
while ( ++counter <= 10 )
    printf( "%u\n", counter );
```

Здесь мы сэкономили одну инструкцию, благодаря тому что увеличение счетчика цикла выполняется непосредственно в условии инструкции `while`, перед его проверкой. Кроме того, в этом коде отпала необходимость в фигурных скобках, окружающих тело инструкции `while`, потому что теперь оно состоит из *единственной* инструкции. Чтобы научиться писать таким сжатым способом, требуется практика. Некоторые программисты считают, что при таком подходе код получается нечитаемым и увеличивается вероятность допустить ошибку.

Значения с плавающей точкой могут быть неточными, поэтому при применении вещественной переменной в качестве счетчика для управления циклом она может получить неточное значение и вызвать логическую ошибку в условном выражении.

Для управления циклами со счетчиками используйте целочисленные значения.

Добавление пустых строк до и после управляющих инструкций и отступов в их телах придает программам структурированный вид и повышает их читаемость.

4.4 Инструкция повторения for

Инструкция `for` автоматически выполняет все, что требуется для организации повторений со счетчиком. Чтобы продемонстрировать ее возможности, перепишем программу из примера 4.1. Результат показан в примере 4.2.

Пример 4.2 | Повторение со счетчиком с помощью инструкции for

```
 1 // Пример 4.2: fig04_02.c
 2 // Повторение со счетчиком с помощью инструкции for.
 3 #include <stdio.h>
 4
 5 // выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    unsigned int counter; // определение счетчика
 9
10    // инициализация, условие повторения и увеличение счетчика
11    // все это включено в заголовок инструкции for.
12    for ( counter = 1; counter <= 10; ++counter ) {
13       printf( "%u\n", counter );
14    } // конец for
15 } // конец функции main
```

Ниже приводится описание работы программы. Когда начинается выполнение инструкции `for`, управляющая переменная инициализируется значением `1`. Затем проверяется условие продолжения цикла `counter <= 10`. Так как начальное значение `counter` равно `1`, условие удовлетворяется и выполняется инструкция `printf` (в строке13), которая выводит значение счетчика, то есть `1`. Затем значение управляющей переменной увеличивается на единицу выражением `++counter` и начинается следующая итерация цикла с проверки условия. Так как управляющая переменная теперь имеет значение `2`, условие снова выполняется, и программа снова вызывает функцию `printf`. Этот процесс продолжается, пока управляющая переменная `counter` не получит значение `11` – проверка условия продолжения цикла в этом случае вернет ложное значение, и повторение прекратится. Программа продолжит выполнение с первой инструкции, следующей за инструкцией `for` (в данном случае это конец программы).

Элементы заголовка инструкции for

На рис. 4.1 изображено строение заголовка инструкции `for`. Обратите внимание, что инструкция `for` содержит «все, что требуется», – она определяет все элементы, необходимые для организации повторения со счетчиком. Если тело инструкции `for` состоит более чем из одной инструкции, его следует заключить в фигурные скобки.

Стандарт языка C позволяет объявить управляющую переменную в разделе инициализации заголовка инструкции `for` (например: `int counter = 1`). Полный пример такого подхода будет показан в приложении E.

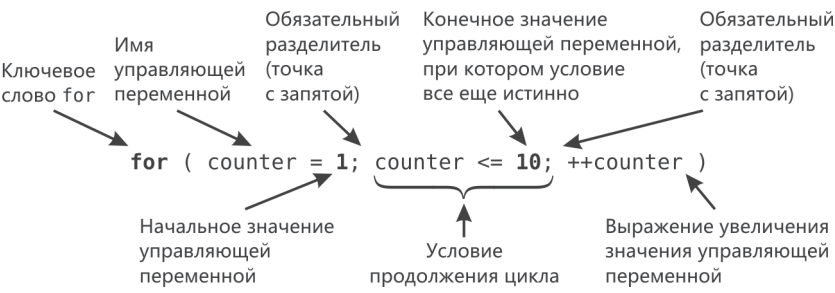


Рис. 4.1 | Компоненты заголовка инструкции for

Ошибка занижения на единицу

Обратите внимание, что в примере 4.2 используется условие продолжения цикла `counter <= 10`. Если по ошибке написать условие `counter < 10`, цикл будет выполняться всего 9 раз. Эта типичная логическая ошибка называется **ошибкой занижения на единицу** (off-by-one error).

Использование конечного значения в условном выражении вкупе с оператором `<=` *поможет избежать ошибок занижения на единицу. Например, для цикла, выводящего значения от 1 до 10, условие продолжения цикла должно иметь вид* `counter <= 10`*, а не* `counter < 11` *или* `counter < 10`*.*

Обобщенный формат инструкции for

В общем виде инструкция `for` имеет следующий формат:

```
for ( expression1; expression2; expression3 ) {
   statement
}
```

где *`expression1`* является выражением инициализации управляющей переменной, *`expression2`* – условием продолжения цикла и *`expression3`* увеличивает значение управляющей переменной в каждой итерации. В большинстве случаев инструкция `for` может быть представлена эквивалентной инструкцией `while`:

```
expression1;
while ( expression2 ) {
   statement
   expression3;
}
```

Из этого правила есть исключения, которые мы рассмотрим в разделе 4.9.

Списки выражений, разделенных запятыми

Часто *expression1* и *expression3* являются списками выражений, разделенных запятыми. В действительности запятая в таких списках является **оператором запятой**, гарантирующим выполнение выражений в списке в порядке слева направо. Типом и значением списка выражений являются тип и значение самого правого выражения в списке. Чаще всего оператор запятой используется в инструкции `for`. Его основное назначение – дать возможность выполнить инициализацию и/или наращивание нескольких переменных. Например, в одной инструкции `for` может быть несколько управляющих переменных, которые необходимо инициализировать и наращивать.

В разделах инициализации и наращивания инструкции `for` *используйте только выражения с участием управляющих переменных. Манипуляции с любыми другими переменными должны выполняться либо перед циклом (если они выполняются только один раз, как инструкции инициализации), либо в теле цикла (если они выполняются в каждой итерации, как инструкции инкремента или декремента).*

Выражения в заголовке инструкции for являются необязательными

Все три выражения в инструкции `for` являются *необязательными*. Если отсутствует выражение *expression2*, по умолчанию *будет полагаться*, что условие всегда *истинно*, в результате чего получится *бесконечный цикл*. Выражение *expression1* можно опустить, если управляющая переменная инициализируется где-то в другом месте программы. Выражение *expression3* можно опустить, если новое значение управляющей переменной вычисляется где-то в теле цикла `for` или когда в этом нет необходимости.

Выражение наращивания действует как самостоятельная инструкция

Выражение наращивания управляющей переменной в инструкции `for` действует как самостоятельная инструкция, выполняемая в конце тела цикла `for`. То есть выражения

```
counter = counter + 1
counter += 1
++counter
counter++
```

являются эквивалентными в контексте раздела инструкции `for`, где производится наращивание управляющей переменной. Некоторые программисты предпочитают использовать форму `counter++`, потому что наращивание происходит *после* выполнения тела цикла и постфиксная форма оператора инкремента выглядит более естественно. Так как наращивание переменной выполняется *не* в составе более крупного выражения, обе формы оператора, постфиксная и префиксная, дают совершенно *одинаковый* эффект. Две точки с запятой в инструкции `for` являются обязательными.

Использование запятых вместо точек с запятой в заголовке инструкции `for` *является синтаксической ошибкой.*

Если справа от заголовка инструкции `for` *поставить точку с запятой, телом инструкции* `for` *станет пустая инструкция. Обычно это приводит к логической ошибке.*

4.5 Инструкция for: замечания

1. Разделы инициализации, условия продолжения и наращивания управляющей переменной могут содержать арифметические выражения. Например, если `x = 2` и `y = 10`, тогда инструкция

```
for ( j = x; j <= 4 * x * y; j += y / x )
```

   будет эквивалентна инструкции:

```
for ( j = 2; j <= 80; j += 5 )
```

2. «Наращивание» может быть отрицательным (то есть может использоваться оператор *декремента*, и тогда цикл фактически будет вести отсчет *в обратном направлении*).
3. Если условие продолжения цикла изначально *ложно*, тело цикла не будет выполнено ни разу. Вместо этого управление будет передано первой инструкции, следующей за инструкцией `for`.
4. Управляющая переменная часто используется в операциях вывода или в вычислениях, в теле цикла, но это необязательно. Нередко она применяется только для управления количеством повторений и нигде не используется в теле цикла.
5. Блок-схема инструкции `for` выглядит так же, как блок-схема инструкции `while`. Например, на рис. 4.2 изображена блок-схема следующей инструкции `for`:

```
for ( counter = 1; counter <= 10; ++counter )
   printf( "%u", counter );
```

Она наглядно демонстрирует, что инициализация выполняется только один раз, а наращивание значения управляющей переменной производится после выполнения тела цикла.

Изменять значение управляющей переменной в теле цикла `for` *допускается, но это может приводить к трудноуловимым ошибкам. Лучше всего этого не делать.*

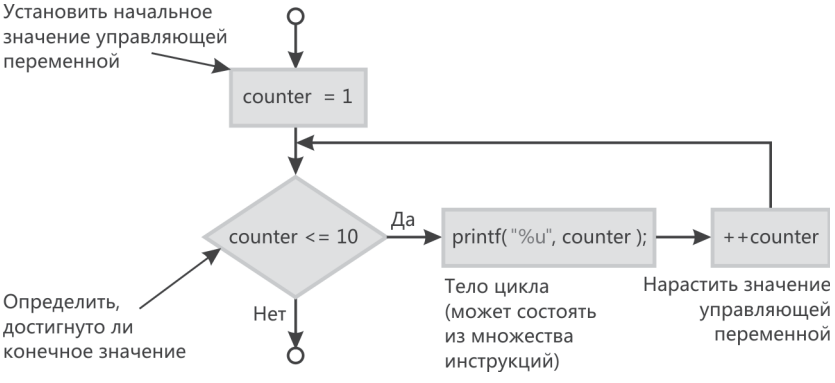


Рис. 4.2 | Блок-схема типичной инструкции повторения for

4.6 Примеры использования инструкции for

Следующие примеры демонстрируют методы изменения значения управляющей переменной в инструкции `for`.

1. Значение управляющей переменной изменяется от 1 до 100 с шагом 1.

```
for ( i = 1; i <= 100; ++i )
```

2. Значение управляющей переменной изменяется от 100 до 1 с шагом –1.

```
for ( i = 100; i >= 1; --i )
```

3. Значение управляющей переменной изменяется от 7 до 77 с шагом 7.

```
for ( i = 7; i <= 77; i += 7 )
```

4. Значение управляющей переменной изменяется от 20 до 2 с шагом –2.

```
for ( i = 20; i >= 2; i -= 2 )
```

5. Управляющая переменная последовательно принимает значения: 2, 5, 8, 11, 14, 17.

```
for ( j = 2; j <= 17; j += 3 )
```

6. Управляющая переменная последовательно принимает значения: 44, 33, 22, 11, 0.

```
for ( j = 44; j >= 0; j -= 11 )
```

Приложение: сумма четных чисел в диапазоне от 2 до 100

В примере 4.3 представлено приложение, использующее инструкцию `for` для вычисления суммы всех четных целых чисел в диапазоне от 2 до 100. В каждой итерации цикла (строки 11–13) значение управляющей переменной `number` прибавляется к значению переменной `sum`.

Пример 4.3 | Вычисление суммы с помощью for

```
 1 // Пример 4.3: fig04_05.c
 2 // Вычисление суммы с помощью for.
 3 #include <stdio.h>
 4
 5 // выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    unsigned int sum = 0; // инициализировать переменную sum
 9    unsigned int number;  // число, прибавляемое к сумме
10
11    for ( number = 2; number <= 100; number += 2 ) {
12       sum += number;    // прибавить число к сумме
13    } // конец for
14
15    printf( "Sum is %u\n", sum ); // вывести сумму
16 } // конец функции main
```

```
Sum is 2550
```

Тело инструкции `for` из примера 4.3 фактически можно было бы вставить в третий раздел ее заголовка:

```
for ( number = 2; number <= 100; sum += number, number += 2 )
   ; // пустая инструкция
```

Инициализацию `sum = 0` также можно было бы вставить в раздел инициализации `for`.

Хотя часто инструкции, предшествующие `for`*, и инструкции, находящиеся в теле цикла, можно вставить в заголовок* `for`*, делать этого не стоит, потому что это ухудшает читаемость программы.*

Старайтесь умещать заголовки управляющих инструкций в одну строку.

Приложение: вычисление сложных процентов

Следующий пример использует инструкцию `for` для вычисления сложных процентов. Рассмотрим формулировку задачи:

> *Некто положил 1000.00 рублей на накопительный счет под 5% годовых. Предполагается, что все накопленные проценты остаются на счете. Требуется вычислить и вывести общую сумму на счете в конце каждого года в течение 10 лет. Вычисления производятся по следующей формуле:*
>
> $a = p(1 + r)^n$,
>
> *где*
>
> *p – первоначальная сумма вклада;*
> *r – процентная ставка;*
> *n – количество лет;*
> *a – сумма вклада в конце n-го года.*

Эта задача решается с применением цикла, в котором вычисляется сумма на счете для каждого из 10 лет. Решение приводится в примере 4.4.

Пример 4.4 | Вычисление сложных процентов

```
 1 // Пример 4.4: fig04_06.c
 2 // Вычисление сложных процентов.
 3 #include <stdio.h>
 4 #include <math.h>
 5
 6 // выполнение программы начинается с функции main
 7 int main( void )
 8 {
 9    double       amount;             // сумма на счете
10    double       principal = 1000.0; // начальная сумма
11    double       rate = .05;         // процентная ставка
12    unsigned int year;               // счетчик лет
13
14    // Вывести заголовки столбцов таблицы
15    printf( "%4s%21s\n", "Year", "Amount on deposit" );
16
17    // вычислить сумму на счете для каждого года из десяти лет
18    for ( year = 1; year <= 10; ++year ) {
19
20       // вычислить новую сумму на счете для указанного года
21       amount = principal * pow( 1.0 + rate, year );
22
23       // вывести строку таблицы
24       printf( "%4u%21.2f\n", year, amount );
25    } // конец for
26 } // конец функции main
```

```
Year   Amount on deposit
 1              1050.00
 2              1102.50
 3              1157.63
 4              1215.51
 5              1276.28
 6              1340.10
 7              1407.10
 8              1477.46
 9              1551.33
10              1628.89
```

Инструкция `for` выполнит тело цикла 10 раз, изменяя значение управляющей переменной от 1 до 10 с шагом 1. Хотя в языке C отсутствует оператор возведения в степень, в стандартной библиотеке имеется функция `pow`, выполняющая эту операцию. Функция `pow(x, y)` возводит значение `x` в степень `y`. Она принимает два аргумента типа `double` и возвращает результат типа `double`. Тип `double` представляет вещественные числа, как тип `float`, с той лишь разницей, что переменные типа `double` обычно могут хранить значения *с намного более высокой точностью*, чем переменные типа `float`. Чтобы получить возможность использовать математические функции, такие как `pow`, необходимо подключить заголовочный файл `<math.h>` (строка 4), без этого программа просто не скомпилируется, потому что компилятор не сможет найти прототип функции `pow`[1]. Функция `pow` требует передать ей два аргумента типа `double`, но переменная `year` является целочисленной. Однако нам не нужно явно выполнять преобразование типа этой переменной, потому что заголовочный файл `math.h` содержит информацию, помогающую компилятору преобразовать целочисленное значение переменной `year` во временное вещественное значение типа `double` *перед* вызовом функции. Эта информация хранится в виде так называемого **прототипа функции** `pow`. Подробнее о прототипах функций рассказывается в главе 5, где также будет представлен список доступных математических функций, в который входит и функция `pow`.

Не используйте типы float и double в финансовых вычислениях

Обратите внимание, что мы объявили переменные `amount`, `principal` и `rate` с типом `double`. Сделано это было лишь по той простой причине, что нам приходится иметь дело с дробными числами.

Не используйте переменные типов `float` и `double` в финансовых вычислениях. Неточность представления вещественных чисел в компьютерах может стать источником неприятных ошибок. Вместо этого вычисления должны выполняться с целыми числами, например с суммами, выраженными в копейках.

---

[1] Во многих системах Linux/UNIX компилятору необходимо также передать флаг `-lm` (например: `gcc -lm fig04_06.c`), чтобы программа из примера 4.4 скомпилировалась. Этот флаг обеспечивает компоновку программы с математической библиотекой.

Ниже приводится простой пример, объясняющий причины ошибок, которые могут возникать при использовании переменных типа `float` или `double` для представления денежных сумм в рублях. Представьте, что в двух переменных типа `float` хранятся суммы в рублях: 14.234 (со спецификатором `%.2f` выводится как 14.23) и 18.673 (со спецификатором `%.2f` выводится как 18.67). При сложении этих переменных получается число 32.907, которое со спецификатором `%.2f` выводится как 32.91. То есть вывод операции сложения на экран мог бы выглядеть так:

```
  14.23
+ 18.67

  32.91
```

Любому понятно, что если сложить видимые на экране числа, получится сумма, равная 32.90! Будьте осторожны!

Форматирование чисел при выводе

Для вывода переменной `amount` в программе используется спецификатор преобразования `%21.2f`. Число `21` в спецификаторе определяет *ширину поля вывода* для данного значения, то есть выведенное значение будет занимать `21` позицию на экране. Число `2` определяет *точность* (то есть количество знаков после запятой). Если длина строки, представляющей число, окажется *меньше* ширины поля, она автоматически будет дополнена пробелами слева для *выравнивания по правому краю поля*. Это особенно удобно для выравнивания вещественных значений с одинаковой точностью (чтобы все десятичные точки выстроились вертикально в ряд). Чтобы обеспечить выравнивание по левому краю, следует добавить `-` (знак «минус») между `%` и шириной поля. Знак минус можно также использовать для выравнивания по левому краю целых чисел (например: `%-6d`) и строк символов (например: `%-8s`). Возможности форматирования функций `printf` и `scanf` мы детально рассмотрим в главе 9.

4.7 Инструкция множественного выбора switch

В главе 3 мы обсудили инструкцию единственного выбора `if` и инструкцию двойного выбора `if...else`. Иногда алгоритм может предусматривать сравнение значения переменной или результата выражения с *множеством вариантов* и предусматривать для каждого из них свое действие. Такую ситуацию называют *множественным выбором*. На этот случай в языке C имеется инструкция множественного выбора `switch`.

Инструкция `switch` состоит из последовательности меток `case`, необязательного варианта `default` и выполняемых инструкций для каждого варианта. В примере 4.5 представлена программа, использующая инструкцию

switch для подсчета количества разных буквенных оценок, полученных учащимися на экзамене.

Пример 4.5 | Подсчет количества разных буквенных оценок с помощью switch

```
// Пример 4.5: fig04_07.c
// Подсчет количества разных буквенных оценок с помощью switch.
#include <stdio.h>

// выполнение программы начинается с функции main
int main( void )
{
   int grade;               // оценка
   unsigned int aCount = 0; // количество оценок A
   unsigned int bCount = 0; // количество оценок B
   unsigned int cCount = 0; // количество оценок C
   unsigned int dCount = 0; // количество оценок D
   unsigned int fCount = 0; // количество оценок F

   puts( "Enter the letter grades." );
   puts( "Enter the EOF character to end input." );

   // цикл, пока пользователь не введет признак конца данных
   while ( ( grade = getchar() ) != EOF) {

      // Определить, какая оценка введена
      switch ( grade ) { // инструкция switch вложена в while

         case 'A':       // буква A в верхнем регистре
         case 'a':       // буква a в нижнем регистре
            ++aCount;    // увеличить aCount
            break;       // необходимо, чтобы выйти из switch

        case 'B':       // буква B в верхнем регистре
        case 'b':       // буква b в нижнем регистре
           ++bCount;    // увеличить bCount
           break;       // необходимо, чтобы выйти из switch

        case 'C':       // буква C в верхнем регистре
        case 'c':       // буква c в нижнемрегистре
           ++cCount;    // увеличить cCount
           break;       // необходимо, чтобы выйти из switch

        case 'D':       // буква D в верхнем регистре
        case 'd':       // буква d в нижнем регистре
           ++dCount;    // увеличить dCount
           break;       // необходимо, чтобы выйти из switch

        case 'F':       // буква F в верхнем регистре
        case 'f':       // буква f в нижнем регистре
           ++fCount;    // увеличить fCount
           break;       // необходимо, чтобы выйти из switch
        case '\n':      // игнорировать символы перевода строки,
        case '\t':      // табуляции
        case ' ':       // и пробелы
```

```
52            break;       // выйти из switch
53
54         default:        // любые другие символы
55            printf( "%s", "Incorrect letter grade entered." );
56            puts( " Enter a new grade." );
57            break;       // необязательно; switch завершится так или иначе
58       } // конец switch
59    } // конец while
60
61    // вывод результатов
62    puts( "\nTotals for each letter grade are:" );
63    printf( "A: %u\n", aCount ); // вывести количество оценок A
64    printf( "B: %u\n", bCount ); // вывести количество оценок B
65    printf( "C: %u\n", cCount ); // вывести количество оценок C
66    printf( "D: %u\n", dCount ); // вывести количество оценок D
67    printf( "F: %u\n", fCount ); // вывести количество оценок F
68 } // конец функции main
```

```
Enter the letter grades.
Enter the EOF character to end input.
a
b
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z
Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Ввод символов

Пользуясь данной программой, пользователь вводит буквенные оценки. В заголовке инструкции `while` (строка 19)

```
19    while ( ( grade = getchar() ) != EOF) {
```

сначала выполняется инструкция присваивания в круглых скобках (`grade = getchar()`). Функция `getchar` (объявлена в заголовочном файле `<stdio.h>`) читает один символ с клавиатуры, который затем сохраняется в целочисленной переменной `grade`. Обычно символы сохраняют в переменных типа `char`. Однако язык C имеет одну важную особенность – символы могут сохраняться в переменных любого целочисленного типа, потому что

в компьютерах они обычно представлены однобайтными целыми числами. То есть мы можем интерпретировать символ и как целое число, и как символ, в зависимости от потребностей. Например, инструкция

```
printf( "The character (%c) has the value %d.\n", 'a', 'a' );
```

использующая спецификаторы преобразования `%c` и `%d`, выведет символ `a` и его целочисленное значение соответственно. В результате на экране появится:

```
The character (a) has the value 97.
```

Число 97 – это числовое представление символа `'a'` в компьютере. Многие современные компьютеры используют для кодировки символов таблицу **ASCII** (**American Standard Code for Information Interchange – американский стандартный код для обмена информацией**), в которой число 97 представляет букву нижнего регистра `'a'`. Список всех ASCII-символов и их десятичные значения можно найти в приложении B. Символы можно также читать с помощью функции `scanf`, используя спецификатор `%c`.

Выражение присваивания само по себе имеет возвращаемое значение. Это то же самое значение, что присваивается переменной слева от оператора `=`. Значением выражения присваивания `grade = getchar()` является символ, возвращаемый функцией `getchar` и присваиваемый переменной `grade`.

Тот факт, что инструкция присваивания возвращает значение, можно использовать для одновременного присваивания одного и того же значения нескольким переменным. Например,

```
a = b = c = 0;
```

первым выполняется присваивание `c = 0` (потому что оператор `=` ассоциативен справа налево). Затем переменной `b` присваивается значение выражения `c = 0` (равное 0). Далее переменной `a` присваивается значение выражения `b = (c = 0)` (также равное 0). В программе значение выражения `grade = getchar()` сравнивается со значением `EOF` (константой, имя которой является сокращением от «end of file» – конец файла). Здесь значение константы `EOF` (обычно равно `-1`) используется в качестве сигнального. Пользователь нажимает системно-зависимую комбинацию клавиш, означающую «конец файла», чтобы сообщить программе «ввод данных завершен». `EOF` – это символическая целочисленная константа, объявленная в заголовочном файле `<stdio.h>` (порядок определения символических констант мы рассмотрим в главе 6). Если значение, присвоенное переменной `grade`, равно `EOF`, программа завершается. Мы выбрали тип `int` для представления символов в программе именно потому, что `EOF` имеет целочисленное значение.

Комбинация клавиш для ввода `EOF` (признака конца файла) зависит от системы.

Сравнение с символической константой вместо –1 повышает переносимость программ. В стандарте C отмечается, что константа `EOF` *должна иметь целочисленное отрицательное значение (необязательно –1). То есть в разных системах константа* `EOF` *может иметь разные значения.*

Ввод признака EOF

В операционных системах Linux/UNIX/Mac OS X признак `EOF` вводится нажатием клавиш

```
<Ctrl> d
```

в отдельной строке. Данная нотация *<Ctrl> d* означает, что сначала следует нажать клавишу *Enter*, а затем одновременно клавиши *Ctrl* и *d*. В других системах, таких как Microsoft Windows, признак `EOF` вводится нажатием комбинации

```
<Ctrl> z
```

В Windows также может потребоваться предварительно нажать клавишу *Enter*.

Пользователь вводит оценки с клавиатуры. После нажатия клавиши *Enter* функция `getchar` читает очередной символ. Если введенный символ не равен `EOF`, выполнение передается инструкции `switch` (строки 22–58).

Инструкция switch

Вслед за ключевым словом `switch` следует имя переменной `grade` в круглых скобках. Круглые скобки с их содержимым называются **управляющим выражением**. Значение этого выражения сравнивается с каждой из **меток** `case`. Допустим, что пользователь ввел букву `C`. Эта буква автоматически сравнивается с каждой меткой `case` в инструкции `switch`. Если совпадение найдено (`case 'C':`), выполняются инструкции в этой ветке `case`. В случае с символом `C` сначала выполняется инструкция увеличения значения переменной `cCount` на 1 (строка 36), а затем инструкция `break`, осуществляющая немедленный выход из инструкции `switch`.

Инструкция `break` передает управление первой инструкции, следующей за инструкцией `switch`. Она используется потому, что после обнаружения совпадения и входа в некоторую ветку `case` выполнение будет продолжено через другие ветки `case` до конца инструкции `switch`, пока не будет встречена инструкция `break`. [Эта особенность редко используется на практике, хотя она прекрасно подходит, чтобы запрограммировать повторяющиеся песни, такие как «The Twelve Days of Christmas»[1]!] Если совпадение ни с одной меткой не будет найдено, выполняются инструкции в метке `default` и выводится сообщение об ошибке.

---

[1] Или про десять негритят. – *Прим. перев.*

Блок-схема инструкции switch

Каждая ветка `case` может содержать одно или более действий. Инструкция `switch` отличается от всех остальных управляющих инструкций тем, что *не* требует использовать фигурные скобки для включения нескольких действий в ветке `case`. В общем представлении инструкция множественного выбора `switch` (с инструкциями `break` в каждой ветке `case`) изображена на рис. 4.3. На блок-схеме видно, что каждая инструкция `break` в конце каждой ветки `case` немедленно передает управление в конец инструкции `switch`.

Отсутствие инструкции break, когда она необходима в инструкции switch, может приводить к логической ошибке.

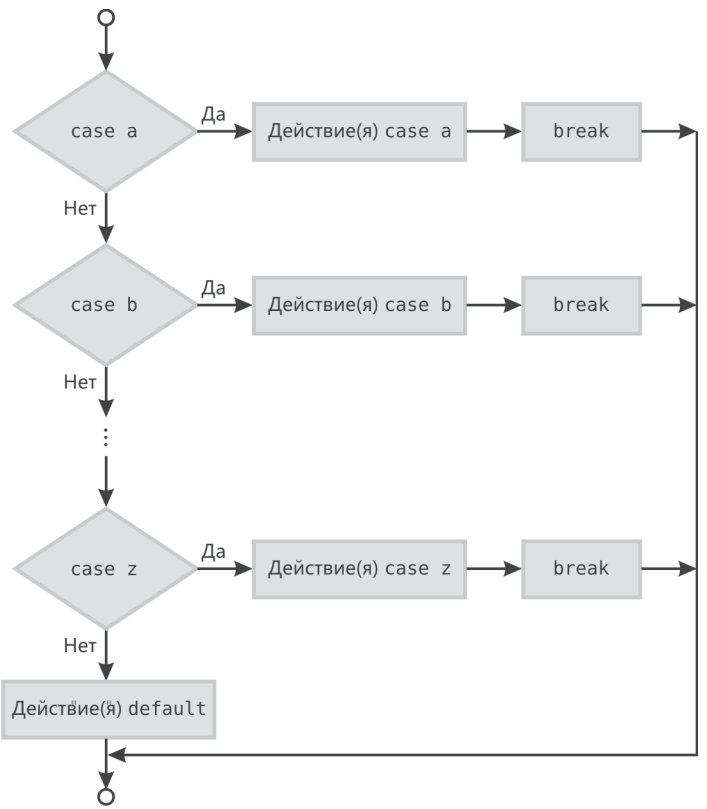


Рис. 4.3 | Инструкция множественного выбора switch с инструкциями break

Добавляйте в инструкции `switch` *ветку* `default`*. Варианты, явно не указанные в метках* `case`*, игнорируются. Ветка* `default` *поможет предотвратить это и предусмотреть обработку исключительных ситуаций. Впрочем, иногда в ветке* `default` *нет необходимости.*

Ветка `default` *может находиться в любом месте в инструкции* `switch`*, однако обычно она помещается в конец.*

Когда ветка `default` *находится в конце инструкции* `switch`*, в нее можно не добавлять инструкцию* `break`*. Впрочем, многие предпочитают добавлять ее для единообразия оформления.*

Пропуск символов перевода строки, табуляции и пробелов

В инструкции `switch` в примере 4.5 строки

```
49          case '\n':      // игнорировать символы перевода строки,
50          case '\t':      // табуляции
51          case ' ':       // и пробелы,
52             break;       // выйти из switch
```

вынуждают программу пропустить символы перевода строки, табуляции и пробелов. Чтение символов по одному может вызывать некоторые проблемы. Чтобы заставить программу читать символы, их нужно посылать компьютеру, нажимая клавишу *Enter*. Однако это вызывает также передачу символа перевода строки вслед за обрабатываемым символом. Часто бывает необходимо предусмотреть специальную обработку символа перевода строки, чтобы обеспечить правильную работу программы. Включив предыдущие метки `case` в нашу инструкцию `switch`, мы предотвратили появление сообщений об ошибках, что выводятся в ветке `default`, при вводе символов перевода строки, табуляции и пробелов.

Не забывайте предусматривать обработку символов перевода строки (и других пробельных символов), когда осуществляете посимвольный ввод.

Две метки, следующие друг за другом (такие как `case 'D': case 'd':` в примере 4.5), просто означают, что для этих вариантов выполняется один и тот же набор действий.

Константные целочисленные выражения

Используя инструкцию `switch`, помните, что значением метки `case` может быть только **константное выражение** – любая комбинация символьных и целочисленных констант, возвращающая целочисленную константу. Символьная константа может быть представлена отдельным символом

в одиночных кавычках (например, `'A'`). Чтобы символы распознавались как константы, они *должны* быть заключены в одиночные кавычки – символы, заключенные в двойные кавычки, распознаются как строки. Целочисленные константы – это обычные целочисленные значения. В нашем примере мы использовали символьные константы. Не забывайте, что символы представлены целыми значениями.

Дополнительно о целочисленных типах

Переносимые языки, такие как C, должны проявлять гибкость в отношении размеров типов данных. Для разных применений могут потребоваться целые числа разных размеров. Язык C поддерживает несколько разновидностей целочисленных типов данных. Вдобавок к типам `int` и `char` в языке C имеются также типы `short int` (имя которого в объявлениях можно сократить до `short`) и `long int` (имя которого в объявлениях можно сократить до `long`). Стандарт языка C определяет минимальный диапазон значений для каждого целочисленного типа, но в некоторых реализациях фактический диапазон может оказаться шире. Для типа `short int` определен минимальный диапазон от –32768 до +32767. Для большинства применений целых чисел вполне достаточно типа `long int`, минимальный диапазон которого простирается от –2147483648 до +2147483647. Диапазон значений типа `int` должен быть не уже диапазона значений типа `short int` и не шире типа диапазона `long int`. На многих современных платформах типы `int` и `long int` представлены одним и тем же диапазоном значений. Тип данных `signed char` можно использовать для представления целых чисел в диапазоне от –128 до +127 или любого символа в компьютере. Полный список целочисленных типов со знаком и без знака можно найти в разделе 5.2.4.2 стандарта языка C.

4.8 Инструкция повторения do…while

Инструкция повторения `do...while` имеет много общего с инструкцией `while`, с той лишь разницей, что в инструкции `while` условие продолжения цикла проверяется в начале цикла, *перед* выполнением тела, а в инструкции `do...while` – *после*. То есть тело инструкции `do...while` гарантированно будет выполнено хотя бы один раз. Когда инструкция `do...while` завершает выполнение, управление передается первой инструкции, следующей за предложением `while`. Инструкция `do...while` не требует использовать фигурные скобки, если ее тело составляет единственная инструкция. Однако многие используют скобки, чтобы не спутать инструкции `while` и `do...while`. Например, строка

```
while ( condition )
```

при беглом взгляде на исходный код обычно воспринимается как заголовок инструкции `while`. Инструкция `do...while` без фигурных скобок вокруг тела с единственной инструкцией выглядит очень похоже:

```
do
   statement
while ( condition );
```

Последнюю строку – `while( condition );` – легко можно принять за инструкцию `while`, тело которой состоит из единственной пустой инструкции. Поэтому, чтобы избежать путаницы, инструкцию `do...while` с телом из единственной инструкции часто записывают так:

```
do {
   statement
} while ( condition );
```

Чтобы устранить потенциальную неоднозначность, можно добавлять фигурные скобки в инструкции `do...while`, даже когда они не требуются.

Когда условие в инструкции повторения никогда не достигает ложного значения, образуется бесконечный цикл. Чтобы предотвратить это, убедитесь в отсутствии точки с запятой непосредственно после `while` или заголовка инструкции. В циклах со счетчиком убедитесь, что управляющая переменная увеличивается (или уменьшается) в цикле. В циклах, управляемых сигнальным значением, убедитесь, что сигнальное значение в конечном итоге подается на вход.

В примере 4.6 инструкция `do...while` используется для вывода чисел от 1 до 10. Управляющая переменная `counter` увеличивается в условии продолжения цикла с помощью префиксного оператора инкремента.

Пример 4.6 | Использование инструкции повторения do...while

```
// Пример 4.6: fig04_09.c
// Использование инструкции повторения do...while.
#include <stdio.h>

// выполнение программы начинается с функции main
int main( void )
{
   unsigned int counter = 1;    // инициализировать счетчик

   do {
      printf( "%u ", counter ); // вывести значение счетчика
   } while ( ++counter <= 10 ); // конец do...while
} // конец функции main
```

```
1 2 3 4 5 6 7 8 9 10
```

Блок-схема инструкции do...while

На рис. 4.4 изображена блок-схема инструкции `do...while`, где видно, что условие проверки продолжения цикла не проверяется, пока действие в теле цикла не будет выполнено хотя бы один раз.

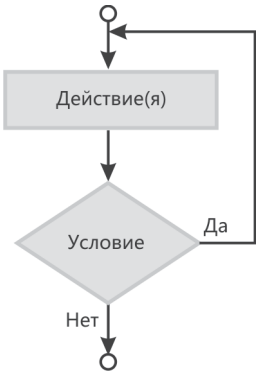


Рис. 4.4 | Блок-схема инструкции do...while

4.9 Инструкции break и continue

Инструкции `break` и `continue` используются для передачи управления. В разделе 4.7 было показано, как использовать инструкцию `break` для выхода из инструкции `switch`. В этом разделе обсуждается применение инструкции `break` в инструкциях повторения.

Инструкция break

Когда `break` встречается внутри инструкции `while`, `for`, `do...while` или `switch`, она *немедленно выходит* из нее. Выполнение программы продолжается с первой инструкции, следующей за инструкцией управления. Обычно инструкция `break` используется для преждевременного прерывания цикла или для выхода из инструкции `switch` (как показано в примере 4.5). В примере 4.7 демонстрируется применение инструкции `break` внутри инструкции `for`. Когда инструкция `if` определяет, что значение `x` становится равным 5, выполняется инструкция `break`. Она завершает инструкцию `for` и передает управление инструкции `printf`, следующей за инструкцией `for`. В результате цикл выполняется только четыре раза.

Пример 4.7 | Использование инструкции break внутри инструкции for

```
1 // Пример 4.7: fig04_11.c
2 // Использование инструкции break внутри инструкции for.
3 #include <stdio.h>
4
5 // выполнение программы начинается с функции main
6 int main( void )
7 {
8    unsigned int x; // счетчик
```

```
 9
10    // повторять 10 раз
11    for ( x = 1; x <= 10; ++x ) {
12
13       // Если значение x равно 5, завершить цикл
14       if ( x == 5 ) {
15          break;           // цикл прерывается, только если x == 5
16       } // конец if
17
18       printf( "%u ", x ); // вывести значение x
19    } // конец for
20
21    printf( "\nBroke out of loop at x == %u\n", x );
22 } // конец функции main
```

```
1 2 3 4
Broke out of loop at x == 5
```

Инструкция continue

Когда инструкция `continue` встречается в инструкции `while`, `for` или `do...while`, она пропускает инструкции, оставшиеся в теле цикла, и переходит в начало следующей итерации. В инструкциях `while` и `do...while` переход осуществляется к выражению проверки условия. В инструкции `for` сначала выполняется выражение наращивания управляющей переменной, а затем выполняется проверка условия продолжения цикла. Выше говорилось, что в большинстве случаев инструкцию `for` можно заменить инструкцией `while`.

Единственное исключение – когда выражение наращивания управляющей переменной в цикле `while` оказывается ниже инструкции `continue`. В этом случае наращивание не выполняется перед проверкой условия, и в результате цикл `while` выполняется не так, как цикл `for`. В примере 4.8 инструкция `continue` используется в инструкции `for`, чтобы пропустить инструкцию `printf` и перейти к следующей итерации.

Пример 4.8 | Использование инструкции continue внутри инструкции for

```
 1 // Пример 4.8: fig04_12.c
 2 // Использование инструкции continue внутри инструкции for.
 3 #include <stdio.h>
 4
 5 // выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    unsigned int x; // счетчик
 9
10    // повторять 10 раз
11    for ( x = 1; x <= 10; ++x ) {
12
13       // если значение x равно 5, перейти к следующей итерации
14       if ( x == 5 ) {
15          continue;        // пропустить оставшиеся инструкции в теле цикла
16       } // конец if
17
```

```
18       printf( "%u ", x ); // вывести значение x
19    } // конец for
20
21    puts( "\nUsed continue to skip printing the value 5" );
22 } // конец функции main
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Некоторые программисты считают, что инструкции `break` *и* `continue` *нарушают принципы структурного программирования. Того же эффекта, что дают эти инструкции, можно добиться за счет применения приемов структурного программирования, с которыми мы вскоре познакомимся, поэтому такие программисты не используют* `break` *и* `continue`.

При правильном использовании инструкции `break` *и* `continue` *действуют быстрее, чем соответствующие им конструкции структурного программирования, с которыми мы вскоре познакомимся.*

Часто приходится идти на компромиссы, выбирая между качеством программного обеспечения и скоростью его работы. Нередко увеличение качества ведет к снижению производительности, и наоборот. Всегда, кроме ситуаций, когда производительность является критически важной, руководствуйтесь следующим правилом: сначала напишите простой и понятный код, а потом сделайте его быстрым и компактным, но только если это действительно необходимо.

4.10 Логические операторы

До сих пор мы видели только простые условия, такие как `counter <= 10`, `total > 1000` и `number != sentinelValue`. Эти условия мы выражали в терминах *операторов отношения* `>`, `<`, `>=`, `<=` и *проверки на равенство* `==` и `!=`. Каждый раз проверялось только *одно* условие. Чтобы принять решение на основе проверки множества условий, нам требовалось выполнить эти проверки в нескольких отдельных или вложенных друг в друга инструкциях `if` или `if...else`. В языке C имеются *логические операторы*, которые можно использовать для конструирования более сложных условий из простых. К логическим относятся операторы `&&` (**логическое И**), `||` (**логическое ИЛИ**) и `!` (**логическое НЕ**, также называется **логическим отрицанием**). Далее мы рассмотрим примеры применения каждого из этих операторов.

Логическое И (&&)

Допустим, прежде чем выполнить некоторое действие или действия, необходимо убедиться в выполнении *сразу двух условий*. В этом случае можно воспользоваться оператором `&&`, как показано ниже:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

Данная инструкция `if` содержит два *простых* условия. В этом примере условие `gender == 1` выясняет, является ли рассматриваемая персона женщиной. Условие `age >= 65` выясняет, является ли рассматриваемая персона пенсионером[1]. Сначала выполняются проверки каждого из условий, потому что операторы `==` и `>=` имеют *более высокий* приоритет, чем `&&`. Затем инструкция `if` определяет результат объединения условий `gender == 1 && age >= 65`, который получает *истинное* значение, только если *оба* условия *истинны*. В итоге, если результат объединения имеет *истинное* значение, счетчик `seniorFemales` увеличивается на 1. Если *какое-либо* из простых условий *ложно*, программа пропускает инструкцию инкремента и переходит к выполнению следующей инструкции.

В табл. 4.1 коротко описывается действие оператора `&&`. В ней перечислены все четыре возможные комбинации значений (нулевых (ложь) и ненулевых (истина)) операндов `expression1` и `expression2`. Такие таблицы часто называют **таблицами истинности**. *Все выражения, включающие операторы отношения, проверки на равенство и/или логические операторы, в языке C возвращают 0 или 1*. Несмотря на то что в C в качестве истинного значения возвращается 1, *любое* ненулевое значение на входе интерпретируется как истинное.

Таблица 4.1 | Таблица истинности оператора логического И (&&)

expression1	expression2	expression1 && expression2
0	0	0
0	Ненулевое значение	0
Ненулевое значение	0	0
Ненулевое значение	Ненулевое значение	1

Логическое ИЛИ (||)

Теперь рассмотрим оператор `||` (логическое ИЛИ). Допустим, прежде чем выполнить некоторое действие или действия, необходимо убедиться в выполнении *любого из двух или обоих* условий. В этом случае можно воспользоваться оператором `||`, как показано ниже:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   puts( "Student grade is A" );
```

Эта инструкция также содержит два простых условия. Условие `semesterAverage >= 90` проверяет, заслуживает ли учащийся оценки «А» за упорный труд в течение семестра. Условие `finalExam >= 90` проверяет, заслуживает ли

[1] В США, где проживает автор книги, граждане выходят на пенсию в возрасте 65 лет независимо от пола. – *Прим. перев.*

учащийся оценки «A» за успешную сдачу заключительного экзамена. Затем инструкция `if` определяет результат объединения условий

```
semesterAverage >= 90 || finalExam >= 90
```

и присваивает учащемуся оценку «A», если выполняется *любое из двух или оба* условия. Сообщение `"Student grade is A"` *не выводится*, только если *оба* простых условия *ложны*. В табл. 4.2 приводится таблица истинности оператора логического ИЛИ (`||`).

Таблица 4.2 | Таблица истинности оператора логического ИЛИ (`||`)

expression1	expression2	expression1 || expression2
0	0	0
0	Ненулевое значение	1
Ненулевое значение	0	1
Ненулевое значение	Ненулевое значение	1

Оператор `&&` имеет *более высокий* приоритет, чем `||`. Оба оператора ассоциативны слева направо. Выражения с операторами `&&` или `||` вычисляются *лишь* до тех пор, пока результат не станет известен однозначно. То есть проверка следующего условия

```
gender == 1 && age >= 65
```

прервется, если выяснится, что значение `gender` не равно `1` (и все выражение целиком будет считаться ложным), и продолжится, если выяснится, что значение `gender` равно `1` (и все выражение целиком будет считаться истинным, только если условие `age >= 65` окажется истинным). Такой способ вычисления логических выражений называется **вычислением по короткой схеме**.

В выражениях с логическим оператором `&&` на место левого операнда ставьте выражения, которые с большей долей вероятности будут принимать ложные значения. В выражениях с логическим оператором `||` на место левого операнда ставьте выражения, которые с большей долей вероятности будут принимать истинные значения. Это поможет увеличить скорость выполнения программы.

Логическое отрицание (!)

В языке C имеется оператор `!` (логическое отрицание), дающий возможность придать условию «противоположный» смысл. В отличие от операторов `&&` и `||`, объединяющих *два* условия (и являющихся, соответственно, *двухместными* операторами), оператор логического отрицания применяется только к *одному* условию (то есть является *унарным* оператором). Оператор логического отрицания помещается перед условием, когда действие или

действия следует выполнить, только если оригинальное условие (без оператора логического отрицания) *ложно*, как, например, в следующем фрагменте:

```
if ( !( grade == sentinelValue ) )
   printf( "The next grade is %f\n", grade );
```

Круглые скобки вокруг условия `grade == sentinelValue` необходимы, так как оператор логического отрицания имеет *более высокий* приоритет, чем оператор проверки на равенство. В табл. 4.3 приводится таблица истинности оператора логического отрицания.

Таблица 4.3 | Таблица истинности оператора логического отрицания (!)

expression1	!expression1
0	0
Ненулевое значение	1

В большинстве случаев можно обойтись без оператора логического отрицания, выразив условие иначе, с помощью другого оператора отношения. Например, предыдущую инструкцию можно переписать так:

```
if ( grade != sentinelValue )
   printf( "The next grade is %f\n", grade );
```

Приоритеты операторов и их ассоциативность

В табл. 4.4 указаны приоритеты и ассоциативность операторов, с которыми мы познакомились к данному моменту. Операторы перечислены сверху вниз, в порядке уменьшения приоритетов.

Таблица 4.4 | Приоритеты и ассоциативность операторов

Операторы	Ассоциативность	Тип
++ (постфиксный) -- (постфиксный)	Справа налево	Постфиксный
+ - ! (тип) ++ (префиксный) - - (префиксный)	Справа налево	Унарный
* / %	Слева направо	Мультипликативный
+ -	Слева направо	Аддитивный
< <= > >=	Слева направо	Проверка отношения
== !=	Слева направо	Проверка на равенство
&&	Слева направо	Логическое И
||	Слева направо	Логическое ИЛИ
?:	Справа налево	Условный
= += -= *= /= %=	Справа налево	Присваивание
,	Слева направо	Запятая

Тип данных _Bool

Стандарт языка С включает определение **логического типа** данных – представлен ключевым словом **`_Bool`**. Переменные этого типа могут хранить

только значение 0 или 1. Не забывайте, что в соответствии с соглашениями в языке C *ложь* и *истину* представляют нулевые и ненулевые значения – значение 0 в условиях интерпретируется как ложь, а ненулевое значение как истина. При попытке присвоить переменной типа `_Bool` ненулевое значение ей в действительности будет присвоено значение 1. Стандарт также включает определение заголовочного файла `<stdbool.h>`, где представляется тип `bool` как синоним типа `_Bool`, а также константы `true` и `false` как именованные представления значений 1 и 0 соответственно. На этапе работы препроцессора идентификаторы `bool`, `true` и `false` замещаются на `_Bool`, 1 и 0. Пример использования `bool`, `true` и `false` можно найти в разделе E.7. В этом примере присутствует функция, определяемая программистом, – с этой возможностью мы познакомимся в главе 5. Вы можете изучить данный пример прямо сейчас или обратиться к нему после прочтения главы 5. Тип данных `_Bool` *не* реализован в Microsoft Visual C++.

4.11 Путаница между операторами равенства (==) и присваивания (=)

Существует одна ошибка, которую программисты допускают настолько часто (независимо от опыта работы), что мы решили посвятить ей отдельный раздел. Эта ошибка заключается в путанице между операторами == (проверки на равенство) и = (присваивания). Самая большая неприятность данной ошибки заключается в том, что обычно она *не обнаруживается компилятором*. Инструкции с этой ошибкой благополучно компилируются и даже выполняются, но приводят к *логическим ошибкам на этапе выполнения*, производя неверные результаты.

Эта проблема обусловлена двумя особенностями языка C. Первая состоит в том, что любое выражение, возвращающее некоторое значение, может использоваться в качестве условия в любой управляющей инструкции. Если выражение возвращает нулевое значение, оно интерпретируется как ложное, а если выражение возвращает ненулевое значение, оно интерпретируется как истинное. Вторая особенность заключается в операторе присваивания, который в языке C возвращает значение – значение, присвоенное переменной слева от оператора присваивания. Например, допустим, что мы хотели написать:

```
if ( payCode == 4 )
   printf( "%s", "You get a bonus!" );
```

а по ошибке написали:

```
if ( payCode = 4 )
   printf( "%s", "You get a bonus!" );
```

Первая инструкция `if` выпишет премию сотруднику, только если его код `payCode` равен 4. Вторая инструкция `if` – содержащая ошибку – выполнит

выражение присваивания в условии. Это выражение просто присвоит переменной `payCode` значение 4. Так как любое ненулевое значение интерпретируется как «истина», условие в данной инструкции `if` будет выполняться всегда, не только когда значение `payCode` равно 4, и сотрудник всегда будет получать премию, независимо от фактического значения `payCode`!

Использование оператора == вместо присваивания и оператора = вместо сравнения на равенство является логической ошибкой.

Левосторонние и правосторонние значения

Вероятно, у вас есть склонность записывать условия, такие как `x == 7`, ставя переменную слева, а константу справа. Поменяв операнды местами так, чтобы константа оказалась слева, а переменная справа, например `7 == x`, вы обезопасите себя от случайной замены оператора == оператором =, так как подобная ошибка будет обнаруживаться компилятором. Компилятор считает такие выражения *синтаксической ошибкой*, потому что слева от оператора присваивания может указываться только имя переменной. Подобным несложным приемом вы предотвратите появление неприятных логических ошибок во время выполнения.

Переменные называют *левосторонними значениями* (lvalues – от «left values», или «левосторонние значения»), потому что они могут использоваться *слева* от оператора присваивания. Константы называют *правосторонними значениями* (rvalues – от «right values», или «правосторонние значения»), потому что они могут использоваться только *справа* от оператора присваивания. *Левосторонние значения* могут также играть роль *правосторонних значений*, но не наоборот.

Когда выражение проверки на равенство содержит переменную, как, например, в `x == 1`, его лучше записать так, чтобы константа стояла слева, а имя переменной справа (`1 == x`), дабы защититься от логических ошибок из-за случайной замены оператора == оператором =.

Путаница операторов = и == в отдельных инструкциях

Оборотная сторона медали ничуть не лучше. Допустим, что нам нужно присвоить некоторое значение переменной с помощью такой простой инструкции, как

```
x = 1;
```

Но по ошибке мы написали:

```
x == 1;
```

Такая инструкция тоже не является синтаксической ошибкой – компилятор просто скомпилирует ее в условное выражение. Если `x` равно 1, выраже-

ние окажется истинным и вернет значение 1. В противном случае оно вернет значение 0. Независимо от того, что вернет выражение, из-за отсутствия оператора присваивания полученное значение просто будет отброшено и значение переменной `x` не изменится, что, вероятно, вызовет логическую ошибку во время выполнения. К сожалению, у нас в арсенале нет приемов, которые помогли бы избавиться от этой проблемы! Однако многие компиляторы выводят предупреждения, встречая подобные инструкции.

4.12 Безопасное программирование на C

Проверка значения, возвращаемого функцией scanf

В примере 4.4 демонстрировалось использование функции `pow` из математической библиотеки, которая возводит первый аргумент в степень, представленную вторым аргументом, и *возвращает* результат в виде значения типа `double`. Результат вычислений затем используется в инструкции, вызвавшей функцию `pow`.

Многие функции возвращают значение, являющееся признаком успешного их выполнения. Например, функция `scanf` возвращает значение типа `int`, которым сообщает об успехе операции ввода. Если в процессе ввода возникла ошибка, `scanf` вернет значение `EOF` (объявлено в `<stdio.h>`); в противном случае она вернет количество прочитанных элементов. Если это значение ***не*** совпадает с ожидаемым количеством элементов, следовательно, функция `scanf` не смогла полностью выполнить операцию ввода.

Взгляните на следующую инструкцию из примера 3.1:

```
20       scanf( "%d", &grade );           // прочитать оценку
```

которая, как предполагается, должна прочитать одно значение типа `int`. Если пользователь введет целое число, `scanf` вернет 1, показывая, что ей действительно удалось прочитать одно значение. Если пользователь введет строку, например `"hello"`, функция `scanf` вернет 0, показывая, что она не смогла прочитать введенную строку как число. В данном случае переменная `grade` не получит значения.

Функция `scanf` может читать сразу несколько значений, например:

```
scanf( "%d%d", &number1, &number2 ); // прочитать два целых числа
```

Если ввод увенчается успехом, `scanf` вернет число 2, указывая, что ей удалось прочитать два значения. Если в качестве первого значения пользователь введет строку, `scanf` вернет 0 и ни одна из переменных, `number1` и `number2`, не получит значения. Если пользователь введет целое число и строку, `scanf` вернет 1 и только переменная `number1` получит значение.

Чтобы повысить надежность операций ввода, проверяйте значение, возвращаемое функцией `scanf`, на равенство ожидаемому количеству элементов. В противном случае программа будет использовать значения пере-

менных, как если бы `scanf` выполнилась успешно. Это может приводить к серьезным логическим ошибкам, вызывающим аварийное завершение программы, или даже к образованию брешей в системе безопасности.

Проверка диапазона

Даже если `scanf` завершится успехом, прочитанные ею значения могут оказаться *недопустимыми*. Например, оценки обычно являются целыми числами в диапазоне 0–100. В программе, осуществляющей ввод таких оценок, необходимо проверять введенные значения на соответствие ожидаемому диапазону значений. В случае ошибки можно предложить пользователю повторить ввод значения. Если в программе требуется организовать ввод значений из определенного множества (например, кодов продукта, не занимающих непрерывный диапазон), можно реализовать проверку вхождения введенных значений в это множество. Более подробно об этом можно прочитать в главе 5 «Integer Security» в книге Роберта Сикорда (Robert Seacord) «Secure Coding in C and C++».

5

Функции

В этой главе вы узнаете:

- как создавать модульные программы из функций;
- как использовать математические функции из стандартной библиотеки C;
- как создавать новые функции;
- как использовать механизмы передачи информации между функциями;
- как действует механизм вызова функций и возврата из них, использующий стек вызовов и кадры стека;
- как пользоваться приемами имитации, основанными на применении генератора случайных чисел;
- как писать и использовать рекурсивные функции – функции, вызывающие самих себя.

5.1 Введение

Как показывает опыт, лучший способ создания больших программ – конструировать их из небольших блоков, или **модулей**, каждый из которых проще в сопровождении, чем оригинальная программа. Этот принцип называется «**разделяй и властвуй**». Данная глава описывает некоторые основные особенности языка C, упрощающие проектирование, реализацию и сопровождение больших программ.

5.2 Модульное программирование на языке C

Модули в языке C называются **функциями**. Обычно программы представляют собой комбинации новых функций, написанных программистом, и функций, входящих в состав стандартной библиотеки C. В данной главе мы рассмотрим обе эти разновидности функций. Стандартная библиотека C содержит богатый набор функций для выполнения математических вычислений, операций со строками и символами, ввода и вывода и множества других полезных операций. Это значительно упрощает работу программиста, потому что данные функции предоставляют широкий круг возможностей.

Обязательно ознакомьтесь с коллекцией функций в стандартной библиотеке C.

Не изобретайте колесо. Всегда, когда это возможно, пользуйтесь функциями из стандартной библиотеки. Это позволит сократить время на разработку.

Использование функций из стандартной библиотеки поможет повысить переносимость программ.

Язык C, как и его стандартная библиотека, является предметом стандарта C и входит в состав стандартных систем программирования на языке C (за исключением библиотек, которые стандартом определяются как необязательные). Функции `printf`, `scanf` и `pow`, использовавшиеся нами в предыдущих главах, являются функциями стандартной библиотеки.

Вы можете писать собственные функции для решения специфических задач в своих программах. Такие функции иногда называют **функциями, определяемыми программистом**. Фактические инструкции, определяющие функцию, пишутся всего один раз и остаются скрытыми от других функций.

Обращение к функциям производится посредством **вызова**. В вызове определяются имя функции и информация (в виде аргументов), необходимая функции для решения стоящей перед ней задачи. Это можно представить себе как иерархию управления предприятием. Начальник (**вызывающая функция**) приказывает подчиненному (**вызываемой функции**) выполнить некоторую работу и сообщить о результатах по завершении (рис. 5.1). Например, функции потребовалось вывести некоторую информацию на экран. Она вызывает функцию-подчиненного `printf` для выполнения этого задания, функция `printf` выводит информацию и сообщает о результатах, или **возвращает управление**, вызывающей функции. Функция-начальник *не* знает, как функция-подчиненный будет выполнять задание. Функция-подчиненный может вызывать другие функции, и начальник никогда не узнает об этом. Далее мы увидим, как такое «сокрытие» технических деталей помогает в разработке программного обеспечения. На рис. 5.1 изображена функция-начальник, раздающая задания нескольким

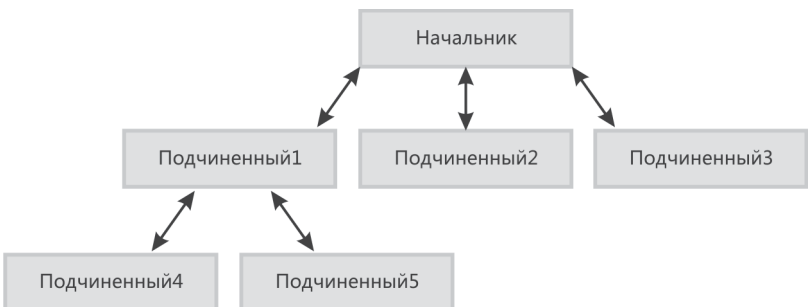


Рис. 5.1 | Иерархические отношения начальник/подчиненный между функциями

функциям-подчиненным с соблюдением иерархии подчинения. Обратите внимание, что `Подчиненный1` действует как начальник для `Подчиненный4` и `Подчиненный5`. Отношения между функциями могут отличаться от иерархии, представленной на рис. 5.1.

5.3 Функции из математической библиотеки

Функции из математической библиотеки позволяют выполнять математические вычисления. Мы будем использовать некоторые из них для знакомства с понятием «функция». Далее в книге мы познакомимся со множеством других функций из стандартной библиотеки.

Обращение к функции в программе обычно заключается в использовании ее имени, за которым следуют круглые скобки со списком **аргументов** в них, разделенных запятыми. Например, вычисление и вывод квадратного корня от числа 900.0 можно выполнить так:

```
printf( "%.2f", sqrt( 900.0 ) );
```

В момент выполнения этой инструкции будет *вызвана* функция `sqrt` из математической библиотеки, которая вычислит квадратный корень от числа в скобках (`900.0`). Число `900.0` здесь – это *аргумент* функции `sqrt`. Предыдущая инструкция должна вывести число `30.00`. Функция `sqrt` принимает аргумент типа `double` и возвращает результат типа `double`. Все функции в математической библиотеке, возвращающие вещественные числа, возвращают результат типа `double`. Обратите внимание, что вывод значений типа `double`, как и значений типа `float`, можно осуществлять с использованием спецификатора преобразования `%f`.

Не забывайте подключать заголовочный файл математической библиотеки директивой препроцессора `#include <math.h>`*, когда используете математические функции.*

Аргументами функций могут быть константы, переменные или выражения. Если предположить, что `c1 = 13.0`, `d = 3.0` и `f = 4.0`, тогда инструкция

```
printf( "%.2f", sqrt( c1 + d * f ) );
```

вычислит и выведет квадратный корень выражения `13.0 + 3.0 * 4.0 = 25.0`, то есть число `5.00`.

В табл. 5.1 приводятся небольшие примеры использования математических функций. Переменные `x` и `y` в этой таблице имеют тип `double`. Стандарт C11 расширил диапазон типов вещественных и комплексных чисел.

Таблица 5.1 | Часто используемые математические функции

Функция	Описание	Пример
`sqrt( x )`	Корень квадратный от *x*	`sqrt( 900.0 )`, результат = `30.0` `sqrt( 9.0 )`, результат = `3.0`
`cbrt( x )`	Корень кубический от *x* (только в стандартах C99 и C11)	`cbrt( 27.0 )`, результат = `3.0` `cbrt( -8.0 )`, результат = `-2.0`
`exp( x )`	Экспоненциальная функция e^x	`exp( 1.0 )`, результат = `2.718282` `exp( 2.0 )`, результат = `7.389056`
`log( x )`	Натуральный логарифм от *x* (по основанию *e*)	`log( 2.718282 )`, результат = `1.0` `log( 7.389056 )`, результат = `2.0`

`log10( x )`	Десятичный логарифм от *x* (по основанию 10)	`log10( 1.0 )`, результат = `0.0`
		`log10( 10.0 )`, результат = `1.0`
		`log10( 100.0 )`, результат = `2.0`

`fabs( x )`	Абсолютное значение *x* как вещественное число	`fabs( 13.5 )`, результат = `13.5`
		`fabs( 0.0 )`, результат = `0.0`
		`fabs( -13.5 )`, результат = `13.5`

`ceil( x )`	Округление до ближайшего меньшего целого, не меньше *x* (округление вверх)	`ceil( 9.2 )`, результат = `10.0` `ceil( -9.8 )`, результат = `-9.0`
`floor( x )`	Округление до ближайшего меньшего целого, не больше *x* (округление вниз)	`floor( 9.2 )`, результат = `9.0` `floor( -9.8 )`, результат = `-10.0`
`pow( x, y )`	*x* в степени *y*	`pow( 2, 7 )`, результат = `128.0` `pow( 9, .5 )`, результат = `3.0`
`fmod( x, y )`	Остаток от деления *x*/*y* как вещественное число	`fmod( 13.657, 2.333 )`, результат = `1.992`
`sin( x )`	Тригонометрический синус от *x* (*x* – в радианах)	`sin( 0.0 )`, результат = `0.0`
`cos( x )`	Тригонометрический косинус от *x* (*x* – в радианах)	`cos( 0.0 )`, результат = `1.0`
`tan( x )`	Тригонометрический тангенс от *x* (*x* – в радианах)	`tan( 0.0 )`, результат = `0.0`

5.4 Функции

Функции позволяют собирать программы из модулей. Все переменные, объявленные в функциях, являются **локальными переменными** – они доступны *только* внутри функций, где они объявлены.

Большинство функций принимают списки параметров, являющиеся средством обмена информацией между функциями. Параметры функции также являются ее локальными переменными.

В программах, содержащих множество функций, функция `main` *зачастую реализована как последовательность вызовов функций, выполняющих основную работу в программе.*

Существует несколько причин «функционализации» программ. Во-первых, следование принципу «*разделяй и властвуй*» делает процесс разработки программ более управляемым. Другой причиной является возможность **повторного использования кода** – использование функций как *строительных блоков* для создания новых программ. Возможность повторного использования кода является одним из основных преимуществ *объектно-ориентированного программирования*, с которым вы познакомитесь, когда будете изучать языки программирования, произошедшие от C, такие как C++, Java и C# (произносится как «си шарп»). При выборе удачных имен и соответствующей реализации функций программы можно создавать из стандартизованных функций, решающих определенные задачи, вместо того чтобы писать специализированный код. Этот прием известен как **абстракция**. Мы пользуемся абстракциями всякий раз, когда обращаемся к функциям из стандартной библиотеки, таким как `printf`, `scanf` и `pow`. Третьей причиной является возможность избежать повторяющегося кода в программах. Заключая код в функции, вы получаете возможность выполнять его в разных частях программы, просто вызывая соответствующую функцию.

Каждая функция должна выполнять какую-то одну, четко ограниченную задачу, а имя функции должно отражать эту задачу. Это упрощает абстрагирование и способствует повторному использованию кода.

Если вы не можете подобрать для функции достаточно краткое имя, отражающее ее назначение, вполне возможно, что она решает слишком много разных задач. Такие функции обычно лучше разбить на несколько функций меньшего размера – иногда этот прием называется «декомпозицией».

5.5 Определение функций

Все программы, представленные нами до сих пор, содержали функцию с именем `main`, вызывающую функции из стандартной библиотеки для решения поставленной задачи. Теперь мы посмотрим, как писать собственные функции. Рассмотрим программу, использующую функцию `square` для вычисления и вывода квадратов целых чисел от 1 до 10 (пример 5.1).

Пример 5.1 | Создание и использование собственной функции

```
1 // Пример 5.1: fig05_03.c
2 // Создание и использование собственной функции.
3 #include <stdio.h>
4
5 int square( int y ); // прототип функции
6
7 // выполнение программы начинается с функции main
8 int main( void )
9 {
```

```
10    int x; // счетчик
11
12    // Повторить 10 раз и в каждой итерации вычислить и вывести квадрат x
13    for ( x = 1; x <= 10; ++x ) {
14       printf( "%d  ", square( x ) ); // вызов функции
15    } // конец for
16
17    puts( "" );
18 } // конец функции main
19
20 // определение функции square, возвращающей квадрат своего параметра
21 int square( int y ) // y - это копия аргумента функции
22 {
23    return y * y;    // вернуть квадрат y как значение типа int
24 } // конец функции square
```

```
1 4 9 16 25 36 49 64 81 100
```

Функция `square` вызывается в функции `main` из инструкции `printf` (строка 14)

```
14       printf( "%d  ", square( x ) ); // вызов функции
```

Она принимает *копию* значения `x` в виде параметра `y` (строка 21). Затем функция `square` вычисляет `y * y`. Результат возвращается функции `printf` в точку вызова `square` (строка 14), а `printf` выводит его. Этот процесс повторяется 10 раз с помощью инструкции `for`.

В определении функции `square` (строки 21–24) видно, что она принимает целочисленный параметр `y`. Ключевое слово `int`, предшествующее имени функции (строка 21), указывает, что `square` возвращает целочисленный результат. Инструкция `return` внутри функции `square` передает результат выражения `y * y` (то есть результат вычислений) обратно вызывающей функции.

Строка 5

```
5 int square( int y ); // прототип функции
```

– это **прототип функции**. Ключевое слово `int` в круглых скобках информирует компилятор, что `square` принимает целочисленное значение. Ключевое слово `int` слева от имени функции `square` сообщает компилятору, что функция возвращает целочисленный результат. С помощью прототипа функции компилятор проверяет все обращения к функции `square` (строка 14) на соответствие *типу возвращаемого значения, правильному количеству аргументов, их типам и указанному порядку их следования*. Подробнее о прототипах функций рассказывается в разделе 5.6.

Определение функции имеет следующий формат:

```
return-value-type function-name( parameter-list )
{
   definitions
   statements
}
```

где имя функции *function-name* – это любой допустимый идентификатор. Тип возвращаемого значения *return-value-type* – тип данных результата, возвращаемого вызывающей функции. Тип `void` возвращаемого значения указывает, что функция *ничего не возвращает*. Все вместе, тип возвращаемого значения *return-value-type*, имя функции *function-name* и список параметров *parameter-list*, иногда называют **заголовком функции**.

Убедитесь, что ваши функции возвращают именно те значения, которые объявлены. Убедитесь, что ваши функции не возвращают значения, которые не объявлены.

Список параметров *parameter-list* – это список элементов, разделенных запятыми, определяющий параметры, передаваемые функции при вызове. Если функция не принимает никаких значений, в качестве списка параметров можно передать ключевое слово `void`. Для каждого параметра в списке явно должен быть указан его тип.

Попытка указать параметры одного типа как `double x, y` вместо `double x, double y` вызовет ошибку на этапе компиляции.

Если в определении функции добавить точку с запятой сразу после закрывающей круглой скобки, завершающей список параметров, это приведет к синтаксической ошибке.

Совпадение имен параметров и локальных переменных вызовет ошибку на этапе компиляции.

Хотя это и допустимо, тем не менее никогда не давайте одинаковые имена аргументам функций и соответствующим им параметрам в определениях функций. Это поможет избежать неоднозначности.

Определения локальных переменных *definitions* и инструкции *statements* внутри фигурных скобок образуют **тело функции**, которое также называется **блоком**. Переменные можно объявлять в любых блоках, а блоки можно вкладывать друг в друга.

Попытка определить функцию внутри другой функции приведет к синтаксической ошибке.

Маленькие функции способствуют многократному использованию кода.

Программы должны создаваться как коллекции маленьких функций. Это упрощает разработку, отладку, сопровождение и модификацию программ.

Функция, принимающая слишком большое количество параметров, вероятно, решает слишком большое количество задач. Подумайте о возможности деления таких функций на более маленькие, решающие отдельные задачи. Заголовок функции должен умещаться в одной строке, если это возможно.

Прототип функции, заголовок функции и вызов функции должны соответствовать друг другу в смысле количества, типов и порядка следования аргументов, а также типа возвращаемого значения.

Существуют три способа вернуть управление из вызываемой функции в точку ее вызова. Если функция ничего *не* возвращает, управление возвращается автоматически, по достижении закрывающей фигурной скобки или выполнением инструкции

```
return;
```

Если функция возвращает результат, вернуть управление в вызывающий код можно с помощью инструкции

```
return expression;
```

Тип значения, возвращаемого функцией main

Обратите внимание, что функция `main` возвращает значение типа `int`. Значение, возвращаемое функцией `main`, используется как признак успешного завершения программы. В прежних версиях языка C требовалось явно возвращать из функции `main` число 0:

```
return 0;
```

чтобы сообщить об успешном выполнении программы. Согласно стандарту языка C, функция `main` должна неявно возвращать 0, если инструкция `return` отсутствует, чем мы и будем пользоваться на протяжении всей книги. Чтобы сообщить о проблеме, возникшей в ходе выполнения программы, можно явно возвращать из `main` ненулевое значение. Дополнительные подробности о том, как сообщить об ошибках, возникших в программе, можно найти в документации с описанием конкретной операционной системы.

Функция maximum

Наш второй пример демонстрирует использование функции `maximum`, определяемой программистом, которая выясняет и возвращает наибольшее из трех целых чисел (пример 5.2). Целые числа вводятся с помощью функции

`scanf` (строка 15). Затем они передаются функции `maximum` (строка 19), которая определяет наибольшее из них и возвращает функции `main` с помощью инструкции `return` (строка 36). После этого возвращаемое значение выводится инструкцией `printf` (строка 19).

Пример 5.2 | Поиск максимального из трех целых чисел

```
 1 // Пример 5.2: fig05_04.c
 2 // Поиск максимального из трех целых чисел.
 3 #include <stdio.h>
 4
 5 int maximum( int x, int y, int z ); // прототип функции
 6
 7 // выполнение программы начинается с функции main
 8 int main( void )
 9 {
10    int number1; // первое целое число, введенное пользователем
11    int number2; // второе целое число, введенное пользователем
12    int number3; // третье целое число, введенное пользователем
13
14    printf( "%s", "Enter three integers: " );
15    scanf( "%d%d%d", &number1, &number2, &number3 );
16
17    // number1, number2 и number3 - это аргументы
18    // для передачи функции maximum
19    printf( "Maximum is: %d\n", );
20 } // конец main
21
22 // Определение функции maximum
23 // x, y и z - это параметры
24 int maximum( int x, int y, int z )
25 {
26    int max = x; // предположить, что x - наибольшее число
27
28    if ( y > max ) { // если y больше, чем max,
29       max = y;      // сохранить y в max
30    } // конец if
31
32    if ( z > max ) { // если z больше, чем max,
33       max = z;      // сохранить z в max
34    } // конец if
35
36    return max;      // max - наибольшее значение
37 } // конец функции maximum
```

```
Enter three integers: 22 85 17
Maximum is: 85
```

```
Enter three integers: 47 32 14
Maximum is: 47
```

```
Enter three integers: 35 8 79
Maximum is: 79
```

5.6 Прототипы функций: обсуждение деталей

Прототипы функций являются важной особенностью языка C. Она была заимствована из C++. Компилятор использует прототипы функций для проверки корректности их вызовов. В ранних версиях C такая проверка *не* выполнялась, поэтому было возможно написать неправильный вызов функции, и компилятор не считал это ошибкой. Подобные вызовы могли приводить к фатальным ошибкам во время выполнения, завершающим программу, и к нефатальным, вызывающим проблемы, трудно поддающиеся диагностике. Добавление поддержки прототипов исправило этот недостаток.

Включайте в текст программы прототипы всех функций, чтобы использовать преимущества механизма проверки типов. Используйте директиву `#include` *препроцессора для подключения стандартных заголовочных файлов с целью получения прототипов функций из стандартной библиотеки и других заголовочных файлов, содержащих прототипы функций, созданных вами и/или вашими коллегами.*

Функция `maximum` из примера 5.2 имеет следующий прототип (строка 5):

```
5 int maximum( int x, int y, int z ); // прототип функции
```

Он сообщает, что `maximum` принимает три аргумента типа `int` и возвращает результат типа `int`. Обратите внимание, что прототип функции совпадает с первой строкой определения функции `maximum`.

Иногда в прототип функций включают имена параметров (мы предпочитаем поступать именно так) для нужд самодокументирования кода. Эти имена игнорируются компилятором.

Отсутствие точки с запятой в конце прототипа функции считается синтаксической ошибкой.

Ошибки компиляции

Несовпадение вызова функции с ее прототипом вызывает ошибку на этапе компиляции. Ошибка также генерируется при несовпадении прототипа и определения функции. Например, если бы в примере 5.2 прототип функции имел вид:

```
void maximum( int x, int y, int z );
```

компилятор сгенерировал бы ошибку из-за несовпадения типа `void` возвращаемого значения в прототипе с типом `int` в заголовке функции.

Приведение типов аргументов и «типичные правила арифметических преобразований»

Еще одной важной особенностью прототипов функций является поддержка **принудительного приведения типов аргументов**. Например, функции `sqrt` из математической библиотеки при вызове можно передать целочисленный аргумент, даже при том, что в прототипе функции, в заголовочном файле `<math.h>`, параметр определен с типом `double`, и функция будет работать правильно. Инструкция

```
printf( "%.3f\n", sqrt( 4 ) );
```

выполнит вызов `sqrt(4)` и выведет `2.000`. Прототип функции вынудит компилятор преобразовать *копию* целого числа 4 в вещественное значение `4.0` перед передачей копии функции `sqrt`. В общем случае, *если значения аргументов не соответствуют в точности типам параметров в прототипе функции, перед вызовом функции они приводятся к соответствующим типам*. Эти преобразования могли бы приводить к ошибочным результатам, если бы компилятор C не использовал **типичные правила арифметических преобразований**. Эти правила определяют порядок преобразований типов значений, чтобы исключить потерю данных. В примере с функцией `sqrt` выше значение типа `int` автоматически преобразуется в значение типа `double` без изменения фактического значения (потому что тип `double` может представлять более широкий диапазон значений, чем тип `int`). Однако обратное преобразование из типа `double` в тип `int` *усекает* дробную часть вещественного числа, то есть изменяет первоначальное значение. Преобразование длинных целых типов в короткие целые (например, из типа `long` в тип `short`) также может приводить к изменению значений.

Типичные правила преобразований автоматически применяются к выражениям, содержащим значения двух типов (их также называют **выражениями смешанного типа**), и обрабатываются компилятором. В выражении смешанного типа компилятор создает временную копию значения, тип которого требуется преобразовать, и затем приводит эту копию к более «широкому» типу, присутствующему в выражении, – оригинальное значение при этом не изменяется. Ниже перечислены типичные правила преобразований для случая, когда в выражении присутствует хотя бы одно вещественное значение:

- если одно из значений имеет тип `long double`, другое также приводится к типу `long double`;
- если одно из значений имеет тип `double`, другое также приводится к типу `double`;
- если одно из значений имеет тип `float`, другое также приводится к типу `float`.

Если выражение смешанного типа содержит только целочисленные типы, применяются аналогичные правила преобразований для целых чи-

сел. В *большинстве* случаев целочисленные типы, расположенные в табл. 5.2 ниже, преобразуются в типы, расположенные выше. Полную информацию об арифметических операндах и типичных правилах преобразования типов можно найти в разделе 6.3.1 стандарта C. В табл. 5.5 перечислены вещественные и целочисленные типы данных с соответствующими им спецификаторами преобразования для функций `printf` и `scanf`.

Таблица 5.5 | Арифметические типы данных и соответствующие им спецификаторы преобразования

Тип данных	Спецификатор для функции printf	Спецификатор для функции scanf
Вещественные типы		
`long double`	`%Lf`	`%Lf`
`double`	`%f`	`%lf`
`float`	`%f`	`%f`
Целочисленные типы		
`unsigned long long int`	`%llu`	`%llu`
`long long int`	`%lld`	`%lld`
`unsigned long int`	`%lu`	`%lu`
`long int`	`%ld`	`%ld`
`unsigned int`	`%u`	`%u`
`int`	`%d`	`%d`
`unsigned short`	`%hu`	`%hu`
`short`	`%hd`	`%hd`
`char`	`%c`	`%c`

Преобразование к более «узким» типам (находящимся в табл. 5.2 ниже) может приводить к получению неверных результатов, поэтому в таких случаях компилятор обычно генерирует предупреждения. Значение может быть приведено к более узкому типу *только* явным присваиванием переменной с более узким типом или с помощью *оператора приведения типа*. Аргументы в вызовах функций приводятся к типам параметров в прототипах функций, как если бы аргументы присваивались переменным этих типов. Если нашу функцию `square` с параметром типа `int` (пример 5.1) вызвать с вещественным аргументом, аргумент будет приведен к типу `int` (более узкому), в результате чего `square` может вернуть неправильное значение. Например, `square(4.5)` вернет `16`, а не `20.25`.

Преобразование из более широкого типа в более узкий может привести к изменению значения. Многие компиляторы в этих случаях выводят предупреждение.

Если прототип функции отсутствует, компилятор сформирует прототип автоматически, на основе первого вхождения – либо определения функции, либо ее вызова. Часто это может приводить к выводу предупреждений и сообщений об ошибках, в зависимости от компилятора.

Всегда подключайте прототипы функций, определяемые вами, или добавляйте их в код программы, чтобы предотвратить ошибки и предупреждения этапа компиляции.

Прототип функции, находящийся за пределами определения какой-либо функции, применяется ко всем вызовам функции, следующим в файле ниже прототипа. Прототип функции, находящийся внутри определения другой функции, применяется только к вызовам внутри этой функции.

5.7 Стек вызовов функций и кадры стека

Чтобы понять, как в языке C происходит вызов функций, необходимо сначала познакомиться с такой структурой данных (коллекцией однородных элементов данных), как **стек**. Стек можно представить как стопку тарелок. Когда тарелка помещается в стопку, она обычно кладется *сверху*. Аналогично, когда требуется извлечь тарелку из стопки, ее обычно снимают *сверху*. Стек также называют структурой данных с организацией элементов **последним пришел, первым вышел** (**Last-In First-Out**, **LIFO**) – *последний* элемент, положенный в стек, будет снят со стека *первым*.

Стек вызовов функций является важнейшим из механизмов, принцип действия которых должен знать и понимать каждый программист. Эта структура данных – действующая «за кулисами» – поддерживает механизм вызова функций и возврата из них. Она также поддерживает создание, использование и удаление автоматических локальных переменных функций. Мы пояснили действие стека на примере стопки с тарелками. Как будет показано ниже, на рис. 5.2–5.4, описание поведения стека выше в *точности* описывает возврат функции после вызова.

Каждая вызываемая функция может вызывать другие функции, которые, в свою очередь, также могут вызывать третьи функции – до возврата из какой-либо функции. Каждая функция рано или поздно должна вернуть управление вызвавшей ее функции. Поэтому необходим некоторый механизм сохранения адреса возврата каждой функции, которая должна будет вернуть управление вызвавшей ее функции. Для этой цели отлично подходит структура данных, которую называют стеком вызовов. Каждый раз, когда одна функция вызывает другую, на стек помещается запись. Эта запись, которая называется **кадром стека**, содержит *адрес возврата*, который необходим вызванной функции, чтобы вернуть управление вызвавшей функции. В кадре стека содержится также некоторая дополнительная информация,

о чем мы поговорим чуть ниже. Когда вызванная функция должна будет вернуть управление, она снимет со стека вызовов функций кадр стека и вернет управление по адресу возврата, указанному в этом кадре.

Каждая вызванная функция *всегда* найдет нужный ей кадр точно на *вершине* стека вызовов. Если вызванная функция вызовет другую функцию, на вершину стека будет помещен кадр стека новой функции. То есть теперь на вершине стека окажется адрес возврата из вновь вызванной функции.

Кадры стека выполняют еще одну важную функцию. Большинство функций имеют *автоматические переменные* – параметры и некоторые или все локальные переменные. Автоматические переменные существуют, пока выполняется функция. Они должны оставаться активными, даже когда функция вызывает другие функции. Но вызываемая функция должна вернуть управление вызывающему коду, автоматические переменные должны «уйти в небытие». Кадр вызываемой функции – отличное место для хранения автоматических переменных. Кадр существует, только пока выполняется владеющая им функция. Когда функция возвращает управление – и автоматические переменные становятся не нужны, кадр стека снимается (или *выталкивается*) со стека, и эти автоматические переменные исчезают.

Конечно, компьютерная память не бесконечна, поэтому для хранения кадров стека может быть отведен вполне определенный объем памяти. Если будет произведено больше вызовов функций, чем сможет вместить стек вызовов, возникнет *фатальная* ошибка, известная как **переполнение стека**.

Стек вызовов в действии

Давайте теперь посмотрим, как стек вызовов поддерживает работу функции `square`, вызываемой из функции `main` (строки 8–13 в примере 5.3). Сначала операционная система вызовет функцию `main` – она поместит на стек кадр этой функции (рис. 5.2). Кадр стека функции `main` сообщает ей, как вернуть управление операционной системе (то есть выполнить переход по адресу возврата `R1`), и содержит достаточно пространства для хранения автоматических переменных функции `main` (то есть переменной `a`, инициализированной значением `10`).

Пример 5.3 | Демонстрация работы стека вызовов и кадров стека с применением функции square

```
 1 // Пример 5.3: fig05_06.c
 2 // Демонстрация работы стека вызовов
 3 // и кадров стека с применением функции square.
 4 #include <stdio.h>
 5
 6 int square( int ); // прототип функции square
 7
 8 int main()
 9 {
10    int a = 10; // значение для возведения в квадрат
11                // (локальная автоматическая переменная)
```

```
12    printf( "%d squared: %d\n", a, square( a )); // вывести квадрат a
13 } // конец main
14
15 // возвращает квадрат целого числа
16 int square( int x ) // x - локальная переменная
17 {
18    return x * x;    // вычислить квадрат и вернуть результат
19 } // конец функции square
```

```
10 squared: 100
```

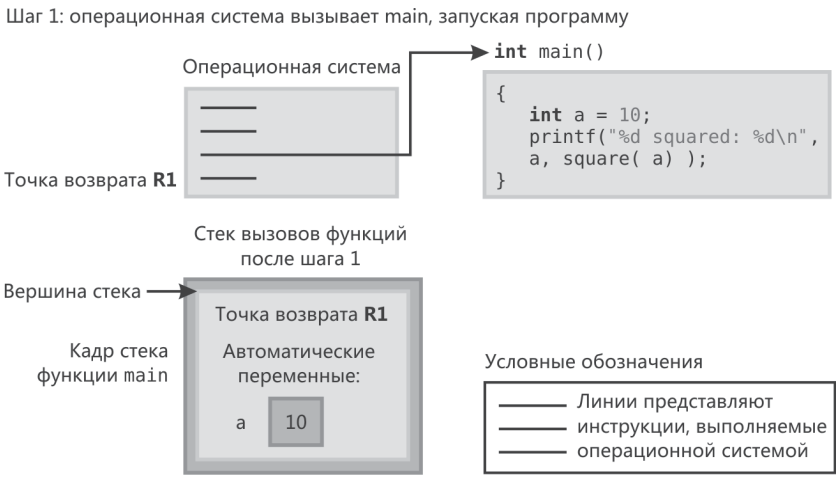


Рис. 5.2 | Стек вызовов функций после того, как операционная система вызовет функцию main программы

Функция `main` – перед возвратом управления операционной системе – вызывает функцию `square` в строке 12 примера 5.3. В результате создается кадр стека для функции `square` (строки 16–19), который помещается на стек вызовов (рис. 5.3). Этот кадр содержит адрес возврата, необходимый функции `square`, чтобы вернуть управление функции (то есть `R2`), и память для хранения автоматической переменной (то есть `x`).

После того как `square` вычислит квадрат своего аргумента, ей необходимо вернуться в функцию `main` и освободить не нужную больше память, занимаемую автоматической переменной `x`. Теперь кадр снимается со стека, что дает функции `square` адрес возврата в `main` (то есть `R2`), и утрачивается связь с автоматической переменной. На рис. 5.4 изображено состояние стека вызовов *после* снятия со стека кадра функции `square`.

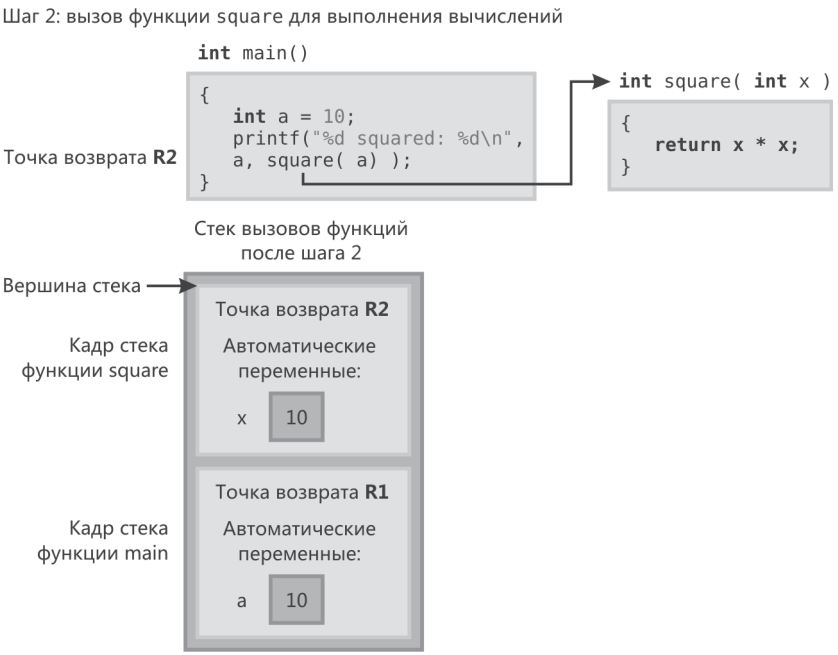


Рис. 5.3 | Стек вызовов функций после того, как main вызовет функцию square для выполнения вычислений

Далее функция `main` выводит результат вызова `square` (строка 12) и достигает закрывающей фигурной скобки, вследствие чего происходит выталкивание кадра стека функции `main`, извлекается адрес возврата в операционную систему (то есть `R1` на рис. 5.2) и утрачивается доступ к автоматической локальной переменной `a`.

Теперь вы знаете, насколько важную роль играет структура данных стека в механизме поддержки выполнения программ. Структуры данных имеют массу применений в программных системах. В главе 12 мы обсудим стеки, очереди, списки, деревья и другие структуры.

5.8 Заголовочные файлы

Каждая стандартная библиотека имеет соответствующий **заголовочный файл**, содержащий прототипы всех функций в этой библиотеке, а также определения различных *типов данных* и констант, необходимых этим функ-

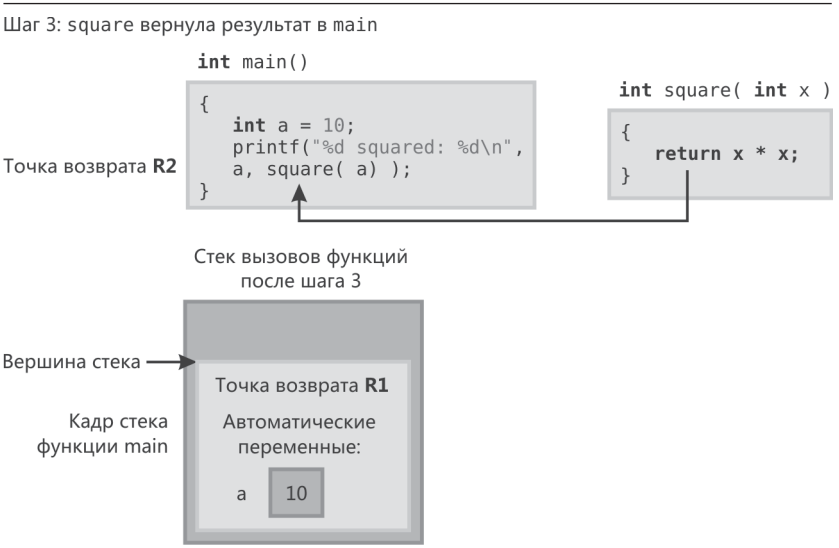


Рис. 5.4 | Стек вызовов функций после того, как square вернет управление функции main

циям. В табл. 5.3 перечислены в алфавитном порядке некоторые заголовочные файлы из стандартной библиотеки, которые могут подключаться программами. Стандарт языка C включает определения некоторых дополнительных заголовочных файлов. Термин «макрос», несколько упоминаемый в табл. 5.3, подробно будет обсуждаться в главе 13.

Вы можете определять собственные заголовочные файлы. Имена таких файлов должны иметь расширение `.h`. Заголовочные файлы, определяемые программистом, могут подключаться директивой `#include` препроцессора. Например, если принять, что прототип функции `square` находится в заголовочном файле `square.h`, мы можем подключить его к нашей программе следующей директивой:

```
#include "square.h"
```

Дополнительную информацию о заголовочных файлах можно найти в разделе 13.2.

Таблица 5.3 | Некоторые заголовочные файлы стандартной библиотеки

Заголовочный файл	Описание
`<assert.h>`	Содержит дополнительную информацию, которая может пригодиться при отладке программ
`<ctype.h>`	Содержит прототипы функций, проверяющих различные характеристики символов, а также прототипы функций для преобразования символов нижнего регистра в верхний регистр и наоборот
`<errno.h>`	Определяет макросы, которые могут пригодиться для вывода сообщений об ошибках
`<float.h>`	Содержит константы, определяющие размеры вещественных чисел в системе
`<limits.h>`	Содержит константы, определяющие размеры целых чисел в системе
`<locale.h>`	Содержит прототипы функций и другую информацию, позволяющие программе действовать с учетом региональных настроек системы. Региональные настройки дают возможность системам обрабатывать различные соглашения по форматированию данных, таких как дата, время, валюта и большие числа
`<math.h>`	Содержит прототипы функций из математической библиотеки
`<setjmp.h>`	Содержит прототипы функций, позволяющих передавать управление в обход обычного механизма вызова/возврата
`<signal.h>`	Содержит прототипы функций и макросы для обработки различных состояний, которые могут возникнуть в ходе выполнения программы
`<stdarg.h>`	Определяет маскросы для обслуживания списка аргументов функции, количество и типы которых заранее неизвестны
`<stddef.h>`	Содержит определения распространенных типов, используемых в вычислениях на языке C
`<stdio.h>`	Содержит прототипы функций из стандартной библиотеки ввода/вывода и информацию, используемую ими
`<stdlib.h>`	Содержит прототипы функций для выполнения преобразований чисел в строки и строк в числа, выделения памяти, получения случайных чисел и других операций
`<string.h>`	Содержит прототипы функций для обработки строк
`<time.h>`	Содержит прототипы функций и типы для работы со значениями времени и даты

5.9 Передача аргументов по значению и по ссылке

Во многих языках программирования поддерживаются два способа передачи аргументов – **по значению** и **по ссылке**. Когда аргумент передается *по значению*, создается *копия* значения аргумента, которая затем передается вызываемой функции. Изменяя копию, функция не оказывает никакого влияния на оригинальное значение. Когда аргумент передается *по ссылке*, вызывающая функция позволяет вызываемой функции *изменить* значение оригинальной переменной.

Передача по значению должна использоваться всегда, когда вызываемая функция не должна изменять значение оригинальной переменной. Этот способ исключает появление **побочных эффектов** по неосторожности, которые сильно тормозят разработку надежных программных систем. Передача по ссылке должна использоваться только при *полном доверии* к вызываемой функции, когда необходимо обеспечить изменение значения оригинальной переменной.

В языке C все аргументы передаются по значению. Как будет показано в главе 7, существует возможность имитировать передачу по ссылке, используя *оператор взятия адреса* и *оператор разыменования*. В главе 6 также будет показано, как аргументы-массивы автоматически передаются по ссылке для поддержки высокой производительности. В главе 7 мы узнаем, что в этом нет никаких противоречий. А пока сосредоточимся на передаче по значению.

5.10 Генератор случайных чисел

Теперь мы предпримем краткий и, надеемся, интересный экскурс в мир игр. В этом и в следующем разделах мы создадим великолепно структурированную игровую программу, состоящую из множества функций. Программа использует большую часть управляющих инструкций, уже изученных нами.

С помощью функции `rand` (прототип определен в заголовочном файле `<stdlib.h>`) из стандартной библиотеки в приложения можно привнести элемент случайности. Взгляните на следующую инструкцию:

```
i = rand();
```

Функция `rand` генерирует целые числа в диапазоне от 0 до `RAND_MAX` (символическая константа, объявленная в заголовочном файле `<stdlib.`

h>). Стандарт языка C требует, чтобы константа RAND_MAX имела значение не менее 32767 – максимальное значение двухбайтного (то есть 16-битного) целого со знаком. Программы в этом разделе тестировались с компилятором Microsoft Visual C++, где константа RAND_MAX имеет значение 32767, и GNU gcc, где константа RAND_MAX имеет значение 2147483647. Если функция rand действительно возвращает *случайные числа*, все числа в диапазоне от 0 до RAND_MAX имеют равные шансы (то есть вероятность) быть выбранными при следующем вызове rand.

Диапазон значений, возвращаемых функцией rand непосредственно, часто отличается от необходимого конкретному приложению. Например, программе, имитирующей бросание монеты, достаточно всего двух значений – 0, обозначающего «решку», и 1, обозначающей «орел». Программе, имитирующей бросание кубика, достаточно случайных чисел в диапазоне от 1 до 6.

Бросание кубика

Для демонстрации возможностей функции rand создадим программу, имитирующую 20 попыток бросания кубика и выводящую результат каждого броска. Прототип функции rand находится в заголовочном файле <stdlib.h>. Вместе с функцией rand мы будем использовать оператор получения остатка от деления (%), как показано ниже:

```
rand() % 6
```

чтобы получить случайные числа в диапазоне от 0 до 5. Этот прием называется **масштабированием**. Число 6 называется **масштабным множителем**. Затем мы сможем **сдвинуть** диапазон чисел, добавляя 1 к предыдущему результату. Вывод примера 5.4 подтверждает, что результаты находятся в диапазоне от 1 до 6 – фактические случайные числа могут отличаться в зависимости от используемого компилятора.

Пример 5.4 | Масштабирование и сдвиг диапазона случайных чисел с помощью выражения 1 + rand() % 6

```
// Пример 5.4: fig05_11.c
// Масштабирование и сдвиг диапазона случайных чисел
// с помощью выражения 1 + rand() % 6.
#include <stdio.h>
#include <stdlib.h>
// выполнение программы начинается с функции main
int main( void )
{
   unsigned int i; // счетчик

   // 20 повторений
   for ( i = 1; i <= 20; ++i ) {

      // получить случайное число от 1 до 6 и вывести его
      printf( "%10d", 1 + rand() % 6 ) );

```

```
      // если счетчик делится на 5, перейти на новую строку
      if ( i % 5 == 0 ) {
         puts( "" );
      } // конец if
   } // конец for
} // конец main
```

```
6         6         5         5         6
5         1         1         5         3
6         6         2         4         2
6         2         3         4         1
```

Бросание кубика 6 000 000 раз

Чтобы показать, что все числа выпадают примерно с равной вероятностью, попробуем сымитировать 6 000 000 бросков кубика с помощью программы из примера 5.5. Каждое целое число в диапазоне от 1 до 6 должно выпасть примерно 1 000 000 раз.

Пример 5.5 | Бросание кубика 6 000 000 раз

```
// Пример 5.5: fig05_12.c
// Бросание кубика 6 000 000 раз.
#include <stdio.h>
#include <stdlib.h>

// выполнение программы начинается с функции main
int main( void )
{
   unsigned int frequency1 = 0; // счетчик выпадений единиц
   unsigned int frequency2 = 0; // счетчик выпадений двоек
   unsigned int frequency3 = 0; // счетчик выпадений троек
   unsigned int frequency4 = 0; // счетчик выпадений четверок
   unsigned int frequency5 = 0; // счетчик выпадений пятерок
   unsigned int frequency6 = 0; // счетчик выпадений шестерок

   unsigned int roll;  // счетчик попыток, значение от 1 до 6000000
   int face;           // текущая попытка, значение от 1 до 6

   // 6 000 000 повторений и суммирование результатов
   for ( roll = 1; roll <= 6000000; ++roll ) {
      face = 1 + rand() % 6; // случайное число в диапазоне от 1 до 6

      // определить выпавшее значение и нарастить соответствующий счетчик
      switch ( face ) {

         case 1: // выпало число 1
            ++frequency1;
            break;

         case 2: // выпало число 2
            ++frequency2;
            break;

         case 3: // выпало число 3
            ++frequency3;
```

```
36             break;
37
38          case 4: // выпало число 4
39             ++frequency4;
40             break;
41
42          case 5: // выпало число 5
43             ++frequency5;
44             break;
45
46          case 6: // выпало число 6
47             ++frequency6;
48             break; // необязательно
49       } // конец switch
50    } // конец for
51
52 // вывод результатов в табличной форме
53 printf( "%s%13s\n", "Face", "Frequency" );
54 printf( "   1%13u\n", frequency1 );
55 printf( "   2%13u\n", frequency2 );
56 printf( "   3%13u\n", frequency3 );
57 printf( "   4%13u\n", frequency4 );
58 printf( "   5%13u\n", frequency5 );
59 printf( "   6%13u\n", frequency6 );
60 } // конец main
```

```
Face     Frequency
1           999702
2          1000823
3           999378
4           998898
5          1000777
6          1000422
```

Как видно из вывода программы, прием масштабирования и сдвига действительно помогают сымитировать бросание игрового кубика. Обратите внимание на использование спецификатора `%s` для вывода строк `"Face"` и `"Frequency"` в заголовках столбцов (строка 53). После того как мы познакомимся с массивами в главе 6, мы покажем, как можно заменить 26-строчную инструкцию `switch` более элегантной однострочной инструкцией.

Рандомизация генератора случайных чисел

Если выполнить программу из примера 5.4, она снова воспроизведет те же результаты:

```
6         6         5         5         6
5         1         1         5         3
6         6         2         4         2
6         2         3         4         1
```

Еще раз обратите внимание, что программа вывела *ту же самую последовательность значений*. Разве можно назвать эти числа *случайными*? Как ни стран-

но, такая *воспроизводимость* результатов является важнейшей характеристикой функции `rand`. При отладке программы она поможет доказать, что внесенные исправления не повлияли на правильную работу программы.

В действительности функция `rand` генерирует **псевдослучайные числа**. Многократные вызовы `rand` будут возвращать последовательность чисел, *кажущихся* случайными. Однако эта последовательность будет повторяться при каждом последующем запуске программы. Как только программа будет отлажена, ее можно модифицировать так, чтобы на каждом запуске она производила *разные* последовательности случайных чисел. Этот прием называется **рандомизацией** и реализуется с помощью функции `srand` из стандартной библиотеки. Функция `srand` принимает целое число без знака и настраивает функцию `rand` на создание другой последовательности целых чисел.

Применение функции `srand` демонстрируется в примере 5.6. Функция `srand` принимает целое число без знака (`unsigned int`). Чтение значения типа `unsigned int` обеспечивает спецификатор преобразования `%u` в вызове функции `scanf`. Прототип функции `srand` находится в заголовочном файле `<stdlib.h>`.

Давайте запустим программу несколько раз и изучим полученные результаты. Обратите внимание, что при каждом запуске программа генерирует разные последовательности случайных чисел, если вводятся *разные* начальные числа.

Обеспечить рандомизацию *без* ввода начального значения можно с помощью следующей инструкции:

```
srand( time( NULL ) );
```

Она читает текущее время в компьютере, которое служит начальным значением. Функция `time` возвращает количество секунд, прошедших с полуночи 1 января 1970 года. Это значение приводится к типу `unsigned int` и используется как начальное значение в генераторе случайных чисел. Прототип функции `time` находится в заголовочном файле `<time.h>`. Подробнее о значении `NULL` рассказывается в главе 7.

Пример 5.6 | Рандомизация программы имитации бросания кубика

```
 1 // Пример 5.6: fig05_13.c
 2 // Рандомизация программы имитации бросания кубика.
 3 #include <stdlib.h>
 4 #include <stdio.h>
 5
 6 // выполнение программы начинается с функции main
 7 int main( void )
 8 {
 9    unsigned int i; // счетчик
10    unsigned int seed; // начальное число для генератора случайных чисел
11
12    printf( "%s", "Enter seed: " );
13    scanf( "%u", &seed ); // спецификатор %u вводит значение unsigned int
```

```
14
15    srand( seed ); // установка начального значения
16
17    // повторить 10 раз
18    for ( i = 1; i <= 10; ++i ) {
19
20       // получить случайное число в диапазоне от 1 до 6 и вывести
21       printf( "%10d", 1 + ( rand() % 6 ) );
22
23       // если счетчик делится на 5, перейти на новую строку
24       if ( i % 5 == 0 ) {
25          puts( "" );
26       } // конец if
27    } // конец for
28 } // конец main
```

```
Enter seed: 67
6         1         4         6         2
1         6         1         6         4
```

```
Enter seed: 867
2         4         6         1         6
1         1         3         6         2
```

```
Enter seed: 67
6         1         4         6         2
1         6         1         6         4
```

Унификация масштабирования и сдвига диапазона случайных чисел

Значения, возвращаемые функцией `rand`, всегда находятся в диапазоне

```
0 ≤ rand() ≤ RAND_MAX
```

Как вы уже знаете, следующая инструкция помогает сымитировать бросание игрового кубика:

```
face = 1 + rand() % 6;
```

Эта инструкция всегда присваивает переменной `face` целое значение (случайное) из диапазона `1 ≤ face ≤ 6`. *Ширина* этого диапазона (то есть количество неповторяющихся чисел, следующих друг за другом) равна 6, а *начальным значением* диапазона является число 1. Исходя из инструкции выше, видно, что ширина диапазона определяется вторым операндом (величина масштаба) оператора получения остатка от деления (то есть 6), а начальное значение диапазона определяется числом (то есть 1), к которому прибавляется результат выражения `rand % 6`. Мы можем сделать операцию получения случайного числа более универсальной:

```
n = a + rand() % b;
```

где `a` – это **величина смещения** (первое число в требуемом диапазоне) и `b` – *масштабный множитель* (ширина желаемого диапазона).

Использовать `srand` *вместо* `rand` *для генерации случайных чисел.*

5.11 Пример: игра в кости

Одной из самых популярных вероятностных игр является игра в кости, в которую играют и в казино, и в глухих переулках по всему миру. Правила игры просты:

> *Игрок бросает два кубика. Каждый кубик имеет шесть граней. На гранях выгравированы 1, 2, 3, 4, 5 и 6 точек. Когда кубики остановятся, подсчитывается сумма точек на гранях кубиков, обращенных вверх. Если в первом броске сумма равна 7 или 11, игрок выигрывает. Если в первом броске сумма равна 2, 3 или 12 (называется «крэпс»), игрок проигрывает. Если в первом броске сумма равна 4, 5, 6, 8, 9 или 10 (называется «пойнт»), игрок продолжает бросать кубики (пытаясь «сделать этот же пойнт»), пока не выпадет сумма 7 (проигрыш) или «пойнт» (выигрыш).*

В примере 5.7 приводится реализация игры в кости, а в примере 5.8 демонстрируется несколько сеансов игры в нее.

Пример 5.7 | Имитация игры в кости

```
// Пример 5.7: fig05_14.c
// Имитация игры в кости.
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // содержит прототип функции time

// перечисление констант, представляющих состояние игры
enum Status { CONTINUE, WON, LOST };

int rollDice( void ); // прототип функции

// выполнение программы начинается с функции main
int main( void )
{
   int sum;     // сумма выпавших очков
   int myPoint; // игрок должен сделать этот пойнт, чтобы выиграть

   enum Status gameStatus; // мог содержать CONTINUE, WON или LOST

   // рандомизировать генератор случайных чисел текущим временем
   srand( time( NULL ) );

   sum = rollDice(); // первый бросок кубиков

```

```
25    // определить состояние игры на основе суммы
26    switch( sum ) {
27
28       // выигрыш в первом броске
29       case 7:               // 7 - победа
30       case 11:              // 11 - победа
31          gameStatus = WON;  // игра завершилась победой
32          break;
33
34       // проигрыш в первом броске
35       case 2:               // 2 - проигрыш
36       case 3:               // 3 - проигрыш
37       case 12:              // 12 - проигрыш
38          gameStatus = LOST; // игра завершилась проигрышем
39          break;
40
41       // запомнить пойнт
42       default:
43          gameStatus = CONTINUE; // игрок должен продолжать бросать
                                   // кубики
44          myPoint = sum;         // запомнить пойнт
45          printf( "Point is %d\n", myPoint );
46          break;                 // необязательно
47    } // конец switch
48
49    // пока игра не завершится
50    while ( CONTINUE == gameStatus ) { // продолжать бросать кубики
51       sum = rollDice();               // бросить кубики
52
53       // определить состояние игры
54       if ( sum == myPoint ) {         // победа при выпадении пойнта
55          gameStatus = WON;            // игра завершилась победой
56       } // конец if
57       else {
58          if ( 7 == sum ) {            // проигрыш при выпадении 7
59             gameStatus = LOST;        // игра завершилась проигрышем
60          } // конец if
61       } // конец else
62    } // конец while
63
64    // вывести сообщение о победе или проигрыше
65    if ( WON == gameStatus ) {         // игрок выиграл?
66       puts( "Player wins" );
67    } // конец if
68    else {                             // игрок проиграл
69       puts( "Player loses" );
70    } // конец else
71 } // конец main
72
73 // имитирует бросок кубиков, вычисляет сумму и выводит ее
74 int rollDice( void )
75 {
76    int die1;    // первый кубик
77    int die2;    // второй кубик
78    int workSum; // сумма
79
80    die1 = 1 + ( rand() % 6 ); // получить первое случайное значение
```

```
81    die2 = 1 + ( rand() % 6 ); // получить второе случайное значение
82    workSum = die1 + die2;     // вычислить сумму
83
84    // вывести результат броска
85    printf( "Player rolled %d + %d = %d\n", die1, die2, workSum );
86    return workSum;            // вернуть сумму
87 } // конец функции rollDice
```

Пример 5.8 | Пример игры в кости

Победа игрока в первом броске

```
Player rolled 5 + 6 = 11
Player wins
```

Победа игрока в серии бросков

```
Player rolled 4 + 1 = 5
Point is 5
Player rolled 6 + 2 = 8
Player rolled 2 + 1 = 3
Player rolled 3 + 2 = 5
Player wins
```

Проигрыш игрока в первом броске

```
Player rolled 1 + 1 = 2
Player loses
```

Проигрыш игрока в серии бросков

```
Player rolled 6 + 4 = 10
Point is 10
Player rolled 3 + 4 = 7
Player loses
```

В соответствии с правилами каждый раз игрок должен бросать *два* кубика. Мы определили функцию `rollDice`, имитирующую бросок кубиков. Она вычисляет сумму выпавших очков и выводит ее. Функция `rollDice` определена *один* раз, но вызывается в *двух* местах в программе (строки 23 и 51). Интересно отметить, что функция `rollDice` не принимает аргументов, и мы отметили этот факт, указав ключевое слово `void` в списке параметров (строка 74). Функция `rollDice` возвращает сумму очков, выпавших на двух кубиках, поэтому тип возвращаемого значения `int` указан в заголовке и в прототипе функции.

Перечисления

Игра, в меру запутанная. Игрок может победить или проиграть в первом броске, а может победить или проиграть в последующих бросках. Переменная `gameStatus` определена с новым типом – `enumStatus` – и предназначена для хранения информации о состоянии игры. В строке `8` объявляется новый

тип, определяемый программистом, который называется **перечислением**. Тип-перечисление объявляется с помощью ключевого слова `enum` и определяет множество целочисленных констант, представленных идентификаторами. Иногда **константы перечислений** называют символическими константами. Первой константе в перечислении присваивается значение 0, а каждой последующей – очередное целое число, больше предыдущего на 1. В строке 8 константа `CONTINUE` получит значение 0, константа `WON` – значение 1, и константа `LOST` – значение 2. Константам в перечислении `enum` можно также присваивать целые числа (см. главу 10). Идентификаторы констант в перечислениях должны быть уникальными, но значения могут повторяться.

Попытка присвоить значение константе перечисления после ее определения вызовет синтаксическую ошибку.

В именах констант перечислений используйте только буквы верхнего регистра, чтобы они выделялись в тексте программы и их сложнее было бы спутать с переменными.

Когда игра завершается выигрышем, первым броском или последующим, переменная `gameStatus` получает значение `WON`. Когда игра завершается проигрышем, `gameStatus` получает значение `LOST`. В противном случае `gameStatus` получает значение `CONTINUE` и игра продолжается.

Окончание игры после первого броска

Если игра завершается после первого броска, инструкция `while` (строки 50–62) не выполняется, потому что переменная `gameStatus` хранит значение, отличное от `CONTINUE`. Далее программа выполнит инструкцию `if...else` в строках 65–70 и выведет `"Player wins"`, если `gameStatus` хранит значение `WON`, или `"Player loses"` в противном случае.

Окончание игры после одного из последующих бросков

Если игра *не* завершилась после первого броска, значение `sum` сохраняется в переменной `myPoint`. Выполнение продолжается в инструкции `while`, потому что переменная `gameStatus` хранит значение `CONTINUE`. В каждом цикле `while` вызывается функция `rollDice`, возвращающая новую сумму `sum` выпавших очков. Если сумма совпадает со значением `myPoint`, в переменную `gameStatus` записывается значение `WON` как признак победы игрока, условие в инструкции `while` становится ложным, инструкция `if...else` выводит `"Player wins"`, и выполнение программы завершается. Если сумма равна 7 (строка 58), в переменную `gameStatus` записывается значение `LOST` как признак проигрыша игрока, условие в инструкции `while` становится ложным, инструкция `if...else` выводит `"Player loses"`, и выполнение программы завершается.

Архитектура управления
Обратите внимание, насколько интересно выглядит архитектура управления выполнением программы. Мы используем две функции, `main` и `rollDice`, инструкции `switch` и `while`, а также вложенные инструкции `if`...`else` и `if`.

5.12 Классы хранения

В главах 2–4 мы использовали идентификаторы в качестве имен переменных. В число атрибутов переменной входят *имя*, *тип*, *размер* и *значение*. В этой главе мы использовали идентификаторы еще и в качестве имен функций, определяемых программистом. Фактически каждый идентификатор в программе имеет еще несколько атрибутов, включая класс хранения, продолжительность хранения, область видимости и связывание.

Язык C поддерживает **спецификаторы классов хранения** `auto`, `register`[1], `extern` и `static`[2]. Идентификатор класса хранения определяет также продолжительность хранения, область видимости и связывание. Под **продолжительностью хранения** идентификатора понимается период времени, в течение которого этот идентификатор *существует в памяти*. Некоторые идентификаторы существуют очень короткий период времени, некоторые многократно создаются и уничтожаются, а иные существуют на протяжении всего времени выполнения программы. Под **областью видимости** подразумевается область программы, *где* данный идентификатор доступен для обращения к нему. Некоторые идентификаторы доступны в любой точке программы, другие – только в ограниченной области. Под **связыванием идентификатора** понимается его доступность в *одном* или во *всех* исходных файлах программы при соответствующем объявлении. В этом разделе обсуждаются классы и продолжительность хранения. В разделе 5.13 мы обсудим области видимости. В главе 14 познакомимся со связыванием идентификаторов и приемами создания программ, состоящих из нескольких файлов с исходными текстами.

Классы хранения можно разделить на **автоматические** и **статические**. Автоматический класс хранения переменной объявляется с помощью ключевого слова `auto`. Автоматические переменные создаются при входе в блок, где они объявляются. Они продолжают существовать, пока данный блок выполняется, и уничтожаются при выходе программы из блока.

Локальные переменные
Автоматический класс хранения могут иметь только переменные. Автоматический класс хранения обычно получают локальные переменные функций

---

[1] Ключевое слово `register` является пережитком прошлого и не должно использоваться.

[2] Новый стандарт языка C добавил еще один спецификатор класса хранения `_Thread_local`, обсуждение которого выходит далеко за рамки этой книги.

(те, что объявляются в списке параметров или в теле функции). Ключевое слово `auto` позволяет явно объявить автоматический класс хранения. Локальные переменные получают автоматический класс *по умолчанию*, поэтому ключевое слово `auto` редко используется на практике. Далее в книге мы будем называть переменные с автоматическим классом хранения просто **автоматическими переменными**.

Автоматический класс хранения обеспечивает экономию памяти, потому что автоматические переменные существуют, только когда они действительно нужны. Они создаются при входе в блок и уничтожаются при выходе из него.

Статический класс хранения

Ключевые слова `extern` и `static` используются в объявлениях имен переменных и функций со *статическим классом хранения*. Идентификаторы со статическим классом хранения существуют на протяжении всего времени выполнения программы до ее завершения. Память для статических переменных выделяется и инициализируется *только один раз, перед тем как программа начнет выполняться*. Имена функций также продолжают существовать вплоть до завершения программы. Однако тот факт, что имена переменных и функций продолжают существовать до самого завершения программы, еще *не* означает, что они доступны из любой точки программы. Продолжительность хранения и область видимости (*где* данное имя может использоваться) – суть разные понятия, как будет показано в разделе 5.13.

Существуют специальные типы идентификаторов со статическим классом хранения: *внешние идентификаторы* (такие как имена глобальных переменных и функций) и локальные переменные, объявленные со спецификатором классом хранения `static`. Имена глобальных переменных и функций по умолчанию получают класс хранения `extern`. Глобальные переменные создаются размещением их объявлений *за пределами* любых функций, и они сохраняют свои значения на протяжении всего времени выполнения программы. Глобальные переменные и функции доступны любым функциям, определения которых следуют за объявлениями этих переменных и функций. Это одна из причин использования прототипов функций. Когда мы подключаем заголовочный файл `stdio.h` в программе, вызывающей функцию `printf`, прототип функции оказывается в начале файла, что обеспечивает ее доступность во всем файле.

Использование глобальных переменных вместо локальных может приводить к неожиданным побочным эффектам, когда функция, которая не должна иметь доступа к переменной, по ошибке или преднамеренно изменит ее. В общем случае глобальных переменных следует избегать, за исключением особых ситуаций с уникальными требованиями (как описывается в главе 14).

Переменные, используемые только в одной функции, должны объявляться как локальные для этой функции.

Локальные переменные, объявленные с ключевым словом `static`, доступны *только* внутри функций, где они определяются, но, в отличие от автоматических переменных, статические локальные переменные *сохраняют* свои значения после выхода из функции. Когда функция будет вызвана в следующий раз, она обнаружит в статической локальной переменной то же значение, которое было оставлено в ней в момент последнего выхода. Следующая инструкция объявляет локальную переменную `count` статической и инициализирует ее значением 1.

```
static int count = 1;
```

Все числовые переменные со статическим классом хранения по умолчанию инициализируются нулями, если явно не будут указаны иные значения.

При применении к внешним идентификаторам ключевые слова `extern` и `static` получают специальное значение. В главе 14 мы обсудим явное применение `extern` и `static` к внешним идентификаторам в программах, состоящих из нескольких файлов с исходным кодом.

5.13 Правила видимости

Область видимости идентификатора – область программы, в которой можно сослаться на данный идентификатор. Например, когда в блоке определяется локальная переменная, сослаться на нее можно *только* в этом блоке ниже объявления или в блоках, вложенных в данный блок. Всего существуют четыре области видимости идентификаторов: область видимости функции, область видимости файла, область видимости блока и область видимости прототипа функции.

Метки (идентификаторы, за которыми следует двоеточие, например `start:`) являются *единственными* идентификаторами с **областью видимости функции**. Метки доступны *везде* внутри функции, где они появляются, но недоступны за пределами тела функции. Метки используются в инструкциях `switch` (метки в предложениях `case`) и `goto` (см. главу 14). Метки относятся к тонкостям реализации, которые функции старательно скрывают друг от друга. Такое сокрытие – более формально называется **сокрытием информации** – является следствием реализации **принципа наименьших привилегий** – фундаментального принципа проектирования программного обеспечения. В контексте приложения этот принцип гласит, что код должен иметь *лишь* тот объем привилегий и круг доступа, которые действительно необходимы для решения поставленной задачи, и ничуть не больше.

Идентификатор, объявленный за пределами какой-либо функции, получает **область видимости файла**. Такой идентификатор будет «известен» (то есть доступен) из любых функций, объявления которых находятся в том же файле, ниже строки с объявлением идентификатора. Глобальные переменные, определения функций и прототипы функций, находящиеся за пределами функций, получают область видимости файла.

Идентификаторы, объявленные внутри блока, получают **область видимости блока**. Область видимости блока заканчивается в конце блока в позиции закрывающей фигурной скобки (}). Локальные переменные, объявленные в начале функции, получают область видимости блока, как и параметры функции, которые считаются локальными переменными функции. *Любой блок может содержать определения переменных.* Когда один блок вложен в другой и идентификатор, объявленный во вложенном блоке, совпадает с идентификатором, объявленным во внешнем блоке, идентификатор во внешнем блоке становится *недоступен*, пока вложенный блок не завершит выполнения. Это означает, что внутри вложенного блока доступен только его собственный локальный идентификатор. Локальные переменные, объявленные с ключевым словом `static`, также получают область видимости блока, даже при том, что они начинают свое существование с момента запуска программы. То есть класс хранения *не* влияет на область видимости идентификатора.

Единственными идентификаторами, получающими **область видимости прототипа функции**, являются идентификаторы в списке параметров внутри прототипа. Как упоминалось выше, прототипы функций *не* требуют наличия имен в списке параметров – обязательными являются только *типы*. Если имена присутствуют в списке параметров в прототипе функции, они *игнорируются* компилятором. Идентификаторы, указанные в прототипе функции, можно использовать в любом месте в программе, не рискуя породить конфликта имен.

Типичной ошибкой является случайное объявление во внутреннем блоке того же идентификатора, что и во внешнем блоке, когда фактически предполагалось использовать идентификатор из внешнего блока.

Избегайте давать локальным переменным имена, которые скроют имена, объявленные во внешних блоках.

В примере 5.9 демонстрируются проблемы, связанные с областями видимости глобальных переменных, автоматических локальных переменных и статических локальных переменных. Глобальная переменная `x` инициализируется значением 1 (строка 9). Эта глобальная переменная оказывается недоступной в блоках или функциях, где объявляется одноименная локальная переменная `x`. В функции `main` определяется локальная переменная `x`, ини-

циализированная значением 5 (строка 14). Функция `main` выводит эту переменную, чтобы показать, что глобальная переменная `x` недоступна в ней. Затем в функции `main` создается новый блок, в котором объявляется еще одна локальная переменная `x`, инициализированная значением 7 (строка 19). Она также выводится, чтобы показать, что переменная `x`, объявленная во внешнем блоке, оказывается недоступной. По завершении блока переменная `x` со значением 7 автоматически уничтожается, и затем выводится локальная переменная `x`, объявленная во внешнем блоке, чтобы показать, что она снова доступна.

Пример 5.9 | Области видимости

```
1 // Пример 5.9: fig05_16.c
2 // Области видимости.
3 #include <stdio.h>
4
5 void useLocal( void );       // прототип функции
6 void useStaticLocal( void ); // прототип функции
7 void useGlobal( void );      // прототип функции
8
9 int x = 1; // глобальная переменная
10
11 // выполнение программы начинается с функции main
12 int main( void )
13 {
14    int x = 5; // локальная переменная в функции main
15
16    printf("local x in outer scope of main is %d\n", x );
17
18    { // начало новой области видимости
19       int x = 7; // локальная переменная в новой области видимости
20
21       printf( "local x in inner scope of main is %d\n", x );
22    } // конец новой области видимости
23
24    printf( "local x in outer scope of main is %d\n", x );
25
26    useLocal();       // useLocal имеет автоматическую локальную
                        // переменную x
27    useStaticLocal(); // useStaticLocal имеет статическую локальную
                        // переменную x
28    useGlobal();      // useGlobal использует глобальную переменную x
29    useLocal();       // useLocal инициализирует автоматическую
                        // локальную переменную x
30    useStaticLocal(); // статическая локальная переменная x сохраняет
                        // значение
31    useGlobal();      // глобальная переменная x также сохраняет значение
32
33    printf( "\nlocal x in main is %d\n", x );
34 } // конец main
35
36 // useLocal инициализирует локальную переменную x в каждом вызове
37 void useLocal( void )
38 {
39    int x = 25; // инициализируется при каждом вызове useLocal
```

```
40
41    printf( "\nlocal x in useLocal is %d after entering useLocal\n", x );
42    ++x;
43    printf( "local x in useLocal is %d before exiting useLocal\n", x );
44 } // конец функции useLocal
45
46 // useStaticLocal инициализирует статическую локальную переменную x при
47 // первом вызове функции; значение x сохраняется между вызовами этой
48 // функции
49 void useStaticLocal( void )
50 {
51    // инициализируется один раз в начале выполнения программы
52    static int x = 50;
53
54    printf( "\nlocal static x is %d on entering useStaticLocal\n", x );
55    ++x;
56    printf( "local static x is %d on exiting useStaticLocal\n", x );
57 } // конец функции useStaticLocal
58
59 // изменяет глобальную переменную x при каждом вызове
60 void useGlobal( void )
61 {
62    printf( "\nglobal x is %d on entering useGlobal\n", x );
63    x *= 10;
64    printf( "global x is %d on exiting useGlobal\n", x );
65 } // конец функции useGlobal
```

```
local x in outer scope of main is 5
local x in inner scope of main is 7
local x in outer scope of main is 5

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x in useLocal is 25 after entering useLocal
local x in useLocal is 26 before exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

В программе определяются три функции, ни одна из которых не имеет аргументов и ничего не возвращает. Функция `useLocal` определяет автоматическую переменную `x` и инициализирует ее значением 25 (строка 39). Когда программа вызывает `useLocal`, значение переменой `x` выводится на экран, за-

тем оно увеличивается на 1 и выводится снова. Каждый раз, когда вызывается эта функция, автоматическая переменная `x` *повторно инициализируется* значением 25. Функция `useStaticLocal` определяет статическую переменную `x` и инициализирует ее значением 50 в строке 52 (не забывайте, что распределение и инициализация статических переменных выполняются *только один раз, на начальном этапе запуска программы*). Локальные переменные, объявленные с ключевым словом `static`, *сохраняют* свои значения даже после выхода из области видимости. Когда программа вызывает `useStaticLocal`, значение переменой `x` выводится на экран, затем оно увеличивается на 1 и выводится снова. При следующем вызове этой функции статическая локальная переменная `x` будет содержать значение 51. Функция `useGlobal` не определяет никаких переменных, поэтому, когда она обращается к переменной с именем `x`, используется глобальная переменная `x` (строка 9). Когда программа вызывает `useGlobal`, значение глобальной переменой `x` выводится на экран, затем оно увеличивается на 10 и выводится снова. При следующем вызове функции `useGlobal` глобальная переменная будет хранить измененное значение 10. В заключение программа вновь выводит локальную переменную `x` в функции `main` (строка 33), чтобы показать, что ни одна из функций не изменила ее значения, потому что все функции ссылаются на переменные в своих областях видимости.

5.14 Рекурсия

Программы, обсуждавшиеся до сих пор, были построены из функций, которые вызывают друг друга в порядке иерархии. Однако при решении некоторых задач бывает удобно иметь функции, вызывающие сами себя. **Рекурсивная функция** – это функция, *вызывающая себя* прямо или косвенно, через вызовы других функций. Рекурсия – очень сложная тема, обсуждаемая на старших курсах изучения информатики. В этом и в следующем разделах мы представим простые примеры использования рекурсии. В данной книге мы часто будем использовать прием рекурсии, особенно в главах 5–8 и 12, а также в приложении E.

Сначала мы познакомимся с теоретическими основами рекурсии, а затем исследуем несколько программ, содержащих рекурсивные функции. Задачи, решаемые с применением рекурсии, имеют много общего. Рекурсивная функция вызывается для решения некоторой задачи. Фактически рекурсивная функция знает решение только для одного простейшего случая, который также называется **базовым случаем**. Если функция вызывается для базового случая, она просто возвращает результат. Если функция вызывается для решения более сложной задачи, она делит задачу на две концептуальные части: с известным решением и с неизвестным решением. Чтобы сделать рекурсию возможной, последняя часть должна напоминать оригинальную задачу, но быть несколько проще или меньше. Поскольку новая

задача выглядит похожей на оригинальную, функция вызывает саму себя, чтобы решить эту новую задачу. Это называется **рекурсивным вызовом**, или **шагом рекурсии**. Шаг рекурсии также подразумевает выполнение оператора `return`, потому что его результат будет объединяться с частью задачи, решение которой уже известно.

Рекурсивные вызовы выполняются, пока оригинальный вызов функции не вернет управление. Шаг рекурсии может привести ко множеству рекурсивных вызовов, так как в каждом из них функция будет продолжать делить задачу на две концептуальные части. Чтобы в конечном итоге рекурсия завершилась, каждый раз функция вызывает себя с упрощенной версией оригинальной задачи. В конечном итоге такое последовательное упрощение *должно привести к базовому случаю*. Когда функция достигнет базового случая, она вернет результат предыдущему вызову, и последовательность возвратов будет выполняться в обратном направлении, пока не будет достигнут первоначальный вызов функции, после чего она вернет результат программе. Это описание выглядит довольно необычно в сравнении с привычными способами решения задач, которые мы видели до сих пор. Может потребоваться немалая практика в создании рекурсивных функций, прежде чем этот процесс будет казаться естественным. Для демонстрации представленных понятий в действии напишем рекурсивную функцию, решающую известную математическую задачу.

Рекурсивное вычисление факториала

Факториал неотрицательного целого числа n, записывается как $n!$ (произносится как «n факториал»), – это произведение

$$n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1,$$

где случаи 1! и 0! равны 1. Например, 5! – это произведение 5 * 4 * 3 * 2 * 1, результат которого равен 120.

Факториал целого числа `number`, большего или равного 0, можно найти с помощью циклического (нерекурсивного) алгоритма:

```
factorial = 1;

for ( counter = number; counter >= 1; --counter )
   factorial *= counter;
```

Рекурсивное определение функции вычисления факториала вытекает из следующего соотношения:

$$n! = n \cdot (n - 1)!$$

Например, очевидно, что значение 5! равно 5 * 4!, как показано ниже:

```
5! = 5 · 4 · 3 · 2 · 1
5! = 5 · (4 · 3 · 2 · 1)
5! = 5 · (4!)
```

Порядок вычисления 5! представлен на рис. 5.5. Рисунок 5.5а показывает, как выполняется последовательность рекурсивных вызовов, пока не будет достигнут базовый случай 1!, решением которого является значение 1, завершающий рекурсию. Рисунок 5.5б показывает, как возвращаются значения из каждого рекурсивного вызова, пока окончательный результат не будет передан вызывающему коду.

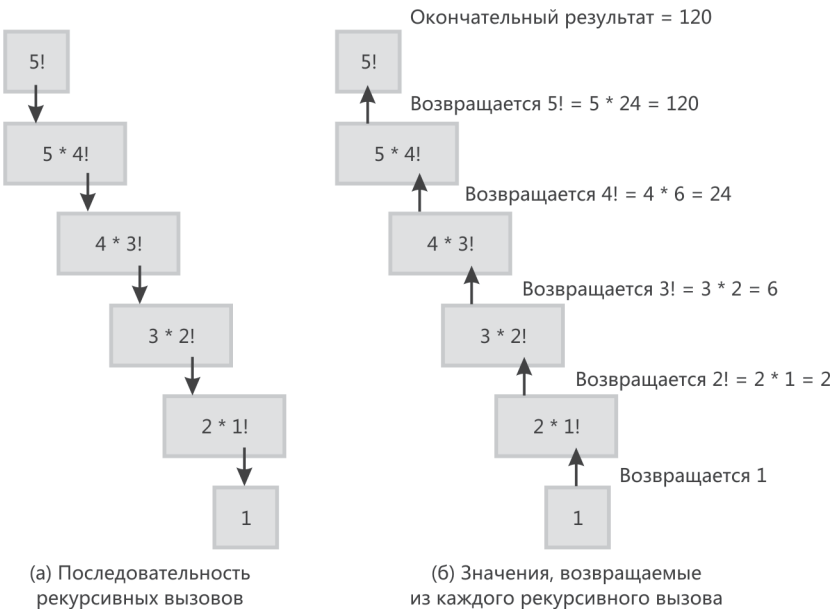


Рис. 5.5 | Рекурсивное вычисление 5!

В примере 5.10 демонстрируется использование рекурсии для вычисления и вывода факториалов целых чисел от 0 до 10 (выбор типа `unsigned long long int` будет обоснован чуть ниже).

Пример 5.10 | Рекурсивная функция вычисления факториала

```
1 // Пример 5.10: fig05_18.c
2 // Рекурсивная функция вычисления факториала.
3 #include <stdio.h>
4
5 unsigned long long int factorial( unsigned int number );
6
7 // выполнение программы начинается с функции main
8 int main( void )
9 {
```

```
10    unsigned int i; // счетчик
11
12    // в каждой итерации вычисляется factorial( i )
13    // и результат выводится на экран
14    for ( i = 0; i <= 21; ++i ) {
15       printf( "%u! = %llu\n", i, factorial( i ) );
16    } // конец for
17 } // конец main
18
19 // определение рекурсивной функции вычисления факториала
20 unsigned long long int factorial( unsigned int number )
21 {
22    // базовый случай
23    if ( number <= 1 ) {
24       return 1;
25    } // конец if
26    else { // шаг рекурсии
27       return ( number * factorial( number - 1 ) );
28    } // конец else
29 } // конец функции factorial
```

```
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
16! = 20922789888000
17! = 355687428096000
18! = 6402373705728000
19! = 121645100408832000
20! = 2432902008176640000
21! = 14197454024290336768
```

Рекурсивная функция сначала проверяет истинность *условия завершения*. Если значение `number` меньше или равно 1, `factorial` возвращает 1, надобность в рекурсии отсутствует, и программа завершается. Если значение `number` больше 1, инструкция

```
return number * factorial( number - 1 );
```

представляет задачу как произведение значения `number` на результат рекурсивного вызова `factorial`, с целью получить факториал значения `number - 1`. Вызов `factorial( number - 1 )` решает немного более простую задачу, чем вызов `factorial( number )`.

Тип unsigned long long int

Функция `factorial` (строки 20–29) принимает значение типа `unsigned int` и возвращает результат типа `unsigned long long int`. Стандарт языка C определяет, что максимальное значение, которое могут хранить переменные типа `unsigned long long int`, не должно быть меньше 18 446 744 073 709 551 615. Как видно из результатов выполнения программы в примере 5.10, значения факториалов растут очень быстро. Мы потому и выбрали тип данных `unsigned long long int`, чтобы программа могла вычислить побольше значений факториалов. Спецификатор преобразования `%llu` используется для вывода значений типа `unsigned long long int`. К сожалению, значения факториалов растут настолько быстро, что использование типа `unsigned long long int` не поможет нам получить достаточно много факториалов до переполнения переменной типа `unsigned long long int`.

Даже используя тип `unsigned long long int`, мы не сможем вычислить факториалы чисел больше 21! Это указывает на недостаток языка C (и большинства других процедурных языков программирования) – его недостаточно простую приспосабливаемость под уникальные требования различных приложений.

Забыть вернуть значение из рекурсивной функции, когда оно необходимо, является ошибкой.

Отсутствие базового случая или ошибка в реализации шага рекурсии, которая не позволит прекратить рекурсивные вызовы по достижении базового случая, приведет к бесконечной рекурсии и в конечном счете к исчерпанию памяти. Эта проблема аналогична проблеме попадания в бесконечный цикл. Бесконечная рекурсия может быть вызвана также вводом недопустимого значения.

5.15 Пример использования рекурсии: числа Фибоначчи

Последовательность чисел Фибоначчи

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```

начинается с 0 и 1, а каждое последующее число является суммой двух предыдущих.

Последовательность чисел Фибоначчи встречается в природе и, в частности, описывает форму спирали. Отношения между соседними числами Фибоначчи стремятся к константе 1.618.... Это число также встречается в природе и называется *золотым сечением*, или *золотой пропорцией*. Форма, в основе построения которой лежит сочетание симметрии и золотого сечения,

способствует наилучшему зрительному восприятию и появлению ощущения красоты и гармонии. Архитекторы часто проектируют окна, комнаты и здания, отношение длины и ширины которых равно золотому сечению. Почтовые открытки также нередко имеют отношение длины и ширины, равное золотому сечению.

Последовательность чисел Фибоначчи можно определить рекурсивно, как показано ниже:

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(n) = fibonacci(n – 1) + fibonacci(n – 2)
```

В примере 5.11 приводится программа, вычисляющая *n*-е число Фибоначчи с помощью рекурсивной функции `ibonacci`. Обратите внимание, что числа Фибоначчи растут очень быстро. Поэтому для параметра функции был выбран тип `unsigned int`, а для возвращаемого значения функции `fibonacci` – тип `unsigned long long int`. Каждая пара строк вывода в примере 5.11 соответствует отдельному запуску программы.

Пример 5.11 | Рекурсивная функция fibonacci

```
// Пример 5.11: fig05_19.c
// Рекурсивная функция fibonacci
#include <stdio.h>

unsigned long long int fibonacci( unsigned int n ); // прототип функции

// выполнение программы начинается с функции main
int main( void )
{
   unsigned long long int result; // число Фибоначчи
   unsigned int number;           // число, введенное пользователем

   // запросить число у пользователя
   printf( "%s", "Enter an integer: " );
   scanf( "%u", &number );

   // вычислить число Фибоначчи с порядковым номером, заданным пользователем
   result = fibonacci( number );

   // вывести число
   printf( "Fibonacci( %u ) = %llu\n", number, result );
} // конец main

// Рекурсивная функция вычисления чисел Фибоначчи
unsigned long long int fibonacci( unsigned int n )
{
   // базовый случай
   if ( 0 == n || 1 == n ) {
      return n;
   } // конец if
   else {    // шаг рекурсии
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
   } // конец else
} // конец функции fibonacci
```

```
Enter an integer: 0
Fibonacci( 0 ) = 0
```

```
Enter an integer: 1
Fibonacci( 1 ) = 1
```

```
Enter an integer: 2
Fibonacci( 2 ) = 1
```

```
Enter an integer: 3
Fibonacci( 3 ) = 2
```

```
Enter an integer: 10
Fibonacci( 10 ) = 55
```

```
Enter an integer: 20
Fibonacci( 20 ) = 6765
```

```
Enter an integer: 30
Fibonacci( 30 ) = 832040
```

```
Enter an integer: 40
Fibonacci( 40 ) = 102334155
```

Вызов `fibonacci` из функции `main` *не* является рекурсивным вызовом (строка 18), но все последующие вызовы – рекурсивные (строка 32). Каждый раз, когда вызывается функция `fibonacci`, она сразу же проверяет наступление базового случая – равенство `n` числу 0 или 1. Если условие выполняется, возвращается значение `n`. Интересно отметить, что если значение `n` больше 1, шаг рекурсии генерирует два рекурсивных вызова, каждый из них решает немного более простую задачу, чем оригинальный вызов `fibonacci`. На рис. 5.6 показано, как функция `fibonacci` вычисляет элемент последовательности с номером 3.

Порядок вычисления операндов

Рисунок 5.6 поднимает одну весьма интересную проблему, связанную с *порядком* вычисления операндов в выражениях, которого придерживаются компиляторы C. Это совершенно иная проблема, отличная от порядка выполнения операторов, который определяется правилами предшествования. На рисунке видно, что вызов `fibonacci(3)` производит *два* рекурсивных вызова: `fibonacci(2)` и `fibonacci(1)`. Но в каком порядке выполняются эти вызовы? Вы можете предположить, что операнды вычисляются справа налево. Для нужд оптимизации C *не* определяет порядок, в каком будут вычисляться операнды большинства операторов (включая `+`). Поэтому вы не должны делать какие-либо предположения о том, в каком порядке будут сделаны вызовы. В действительности первым может быть сделан вы-

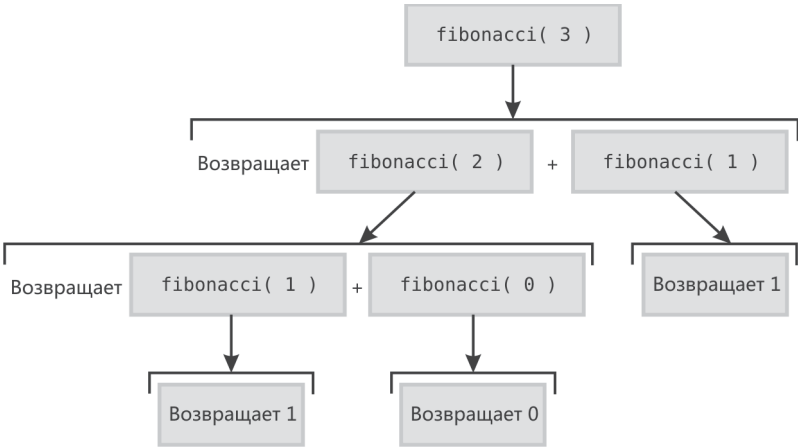


Рис. 5.6 | Последовательность рекурсивных вызовов при вызове fibonacci(3)

зов `fibonacci(2)`, а затем `fibonacci(1)`, или наоборот, сначала `fibonacci(1)`, а затем `fibonacci(2)`. В этой и в большинстве других программ окончательный результат останется тем же. Но в некоторых программах порядок вычисления операндов *может влиять* на окончательный результат. В языке С порядок вычисления операндов четко определен *только для четырех* операторов – `&&`, `||`, запятая (`,`) и `?:`. Первые три являются двухместными операторами и гарантируют вычисление операндов *слева направо*. [Обратите внимание: запятые, разделяющие аргументы в вызовах функций, *не* являются операторами запятой.] Последний оператор – единственный *трехместный* (или *тернарный*) оператор в языке С. Самый левый операнд этого оператора *всегда* вычисляется *первым*; если самый левый операнд не равен нулю, далее вычисляется средний операнд, а последний – игнорируется; если самый левый операнд равен нулю, следующим вычисляется третий операнд, а средний – игнорируется.

Создание программ, полагающихся на какой-то определенный порядок вычисления операндов, кроме операторов `&&`, `||`, `?:` и оператора запятой (`,`), может приводить к ошибкам, потому что компиляторы не всегда вычисляют операнды в том порядке, в каком вы ожидаете.

Программы, правильность работы которых зависит от порядка вычисления операндов, кроме операторов `&&`, `||`, `?:` и оператора запятой (`,`), могут выполняться по-разному, будучи скомпилированными разными компиляторами.

Экспоненциальная сложность

Следует сказать несколько слов предостережений по поводу программ, подобных той, что мы написали выше для получения чисел Фибоначчи. На каждом уровне рекурсии функции `fibonacci` количество вызовов функции *удваивается* – количество рекурсивных вызовов для *n*-го числа Фибоначчи равно 2^n. При таком нарастании процесс вычислений быстро может выйти из-под контроля. Для вычисления всего лишь 20-го числа Фибоначчи требуется выполнить что-то около 2^{20}, или около миллиона вызовов, для вычисления 30-го числа Фибоначчи потребуется порядка 2^{30}, или около миллиарда вызовов, и т. д. Программисты называют такие алгоритмы алгоритмами с экспоненциальной сложностью. Подобные алгоритмы способны «поставить на колени» даже самые мощные компьютеры!

В примере, представленном в этом разделе, было показано наиболее привлекательное на первый взгляд решение задачи вычисления чисел Фибоначчи, однако существуют другие, более удачные решения.

5.16 Рекурсия и итерации

В предыдущих разделах мы рассмотрели две функции, которые легко можно реализовать, используя любой из подходов, рекурсивный или итеративный. В этом разделе мы сравним эти два подхода и узнаем, в каких ситуациях следует отдавать предпочтение тому или иному решению.

- Оба решения, итеративное и рекурсивное, опираются на использование *управляющей структуры*: в итеративном решении используется *структура повторения*, а в рекурсивном – *структура выбора*.
- В обоих решениях, итеративном и рекурсивном, выполняются *повторения*: в итеративном решении явно используется *инструкция повторения*, а в рекурсивном повторения достигаются за счет *повторяющихся вызовов функции*.
- В обоих решениях, итеративном и рекурсивном, выполняется *проверка условия завершения*: при итеративном подходе вычисления заканчиваются, когда *условие продолжения цикла становится ложным*, а в рекурсивном – *по достижении базового случая*.
- В итеративном решении используется повторение со счетчиком, а в рекурсивном выполняется пошаговое приближение к завершению: в итеративном решении постоянно продолжает изменяться счетчик, пока он не достигнет значения, при котором *условие продолжения цикла станет ложным*, а в рекурсивном продолжают решаться все более простые задачи, пока не будет *достигнут базовый случай*.
- Оба решения, итеративное и рекурсивное, могут попасть в *бесконечный цикл*: в итеративном решении цикл становится *бесконечным*, если

условие продолжения *никогда* не станет ложным, а *рекурсия* становится *бесконечной*, если каждый шаг рекурсии *не* упрощает задачу так, что в конечном итоге она *сходится к базовому случаю*.

Рекурсивное решение имеет множество отрицательных сторон. Оно *многократно* использует *механизм вызова функции* и страдает от сопутствующих *накладных расходов*. На решение задачи рекурсивным способом может тратиться слишком много памяти и процессорного времени. Каждый рекурсивный вызов создает еще одну копию функции (в действительности – только переменные функции). Это может приводить к *значительному расходованию памяти*. Итеративное решение обычно лишено недостатков, связанных с накладными расходами на рекурсивные вызовы функции и большим потреблением памяти. Тогда почему иногда предпочтительнее выбирать рекурсию?

Любая задача, которая может быть решена рекурсивно, может быть решена и итеративно (нерекурсивно). Предпочтение рекурсивному подходу обычно отдается, когда он наиболее естественно отражает решение задачи, в результате чего упрощаются отладка и сопровождение программы. Другая причина в пользу выбора рекурсивного решения – итеративное решение может оказаться неочевидным.

Большинство книг по программированию представляет рекурсию в более поздних главах, чем в этой книге. Мы считаем, что рекурсия – достаточно многогранная и сложная тема и что ее лучше представить как можно раньше, чтобы иметь возможность показать как можно больше примеров ее использования в оставшейся части книги. Давайте закроем эту главу несколькими наблюдениями, к которым мы неоднократно будем возвращаться на протяжении всей книги. Простота и ясность программной архитектуры играют важную роль. Не менее важную роль играет высокая производительность. К сожалению, эти две цели часто противоречат друг другу. Простота и ясность программной архитектуры делают более управляемой задачу разработки крупных и сложных систем. Высокая производительность является ключом к реализации систем будущего, которые все больше вычислительных задач будут перекладывать на аппаратное обеспечение. Насколько хорошо согласуются функции с вышесказанным?

Разбиение программ на функции способствует их упрощению. Но все хорошо в меру. Слишком дробное разделение – в сравнении с монолитной (то есть единым блоком) реализацией программ без функций – приведет к увеличению количества вызовов функций и, соответственно, к расходованию процессорного времени на их выполнение. Но хотя монолитные программы могут показывать лучшую производительность, они более сложны в разработке, тестировании, отладке, сопровождении и дальнейшем развитии.

Современные аппаратные архитектуры проектируются так, чтобы обеспечить максимальную эффективность вызовов функций, и нынешние процессоры показывают невероятную скорость работы. Для подавляющего большинства приложений и программных систем, которые вам придется создавать, простота и ясность будут иметь намного большее значение, чем решение проблем с производительностью. Однако во многих приложениях и системах, которые пишутся на языке C, таких как игры, системы реального времени, операционные системы и встраиваемые системы, производительность имеет критическое значение, поэтому мы будем знакомить вас с советами по повышению производительности на протяжении всей книги.

5.17 Безопасное программирование на C

Случайные числа и безопасность

В разделе 5.10 мы познакомились с функцией `rand`, возвращающей псевдослучайные числа. Стандартная библиотека C не реализует безопасный генератор случайных чисел. Согласно описанию в стандарте языка C, функция `rand`: «не дает никаких гарантий качества последовательности случайных чисел, а некоторые реализации *печально известны* неслучайным характером установки младших битов». В рекомендации MSC30-C центра CERT указывается, что функции, генерирующие псевдослучайные числа, должны гарантировать *непредсказуемость* возвращаемых ими чисел – это чрезвычайно важно, например, в криптографии и для организации безопасности приложений. Рекомендация представляет несколько небольших генераторов случайных чисел, которые могут считаться безопасными. Например, Microsoft Windows предоставляет функцию `CryptGenRandom`, а POSIX-совместимые системы (такие как Linux) предоставляют функцию `random`, дающую более непредсказуемые (а значит, и более безопасные) результаты. За дополнительной информацией обращайтесь к тексту рекомендации MSC30-C по адресу: `https://www.securecoding.cert.org`.

Если вы занимаетесь созданием промышленных приложений, требующих применения случайных чисел, вам необходимо исследовать функции, рекомендованные для вашей платформы.

6

Массивы

В этой главе вы научитесь:

- использовать массивы для представления списков и таблиц значений;
- определять массивы, инициализировать их и обращаться к отдельным элементам массивов;
- определять символические константы;
- передавать массивы функциям;
- использовать массивы для хранения, сортировки и организации поиска значений в списках и таблицах;
- определять и использовать многомерные массивы.

6.1 Введение

Эта глава служит введением в структуры данных. **Массивы** – это структуры данных, состоящие из взаимосвязанных элементов данных одного типа. В главе 10 мы обсудим понятие структуры (`struct`) в языке C – это структуры данных, состоящие из взаимосвязанных элементов данных, возможно разных типов. Массивы и структуры являются «статическими» сущностями в том смысле, что сохраняют свой размер в ходе выполнения программы (разумеется, они могут иметь автоматический класс хранения и, соответственно, создаваться и уничтожаться при входе в блок и выходе из него).

6.2 Массивы

Массив – это группа элементов одного типа, хранящихся в *непрерывной* области памяти. Чтобы обратиться к конкретному элементу массива, необходимо указать *имя массива* и порядковый номер элемента в массиве.

На рис. 6.1 изображен массив целых чисел с именем `c`, содержащий 12 элементов. К любому из этих элементов можно обратиться, указав имя массива и *порядковый номер* элемента в квадратных скобках (`[]`). Первый элемент любого массива имеет порядковый номер **0**. Имя массива, как и имя любой другой переменной, может состоять только из букв, цифр и символов подчеркивания и не может начинаться с цифры.

Порядковый номер в квадратных скобках называется **индексом**. Индексом может быть только целое число или целочисленное выражение. Например, если предположить, что `a = 5` и `b = 6`, инструкция

```
c[ a + b ] += 2;
```

прибавит число 2 к элементу массива `c[11]`. Имя массива с индексом является *левосторонним выражением* (lvalue), которое можно использовать слева от оператора присваивания.

Рассмотрим массив на рис. 6.1 поближе. Массив имеет **имя** `c`. Его 12 элементов доступны как `c[0]`, `c[1]`, `c[2]`, ..., `c[10]` и `c[11]`. В элементе `c[0]` хранится значение 45, в элементе `c[1]` – значение 6, в элементе `c[2]` – значение 0, в элементе c[7] – значение 62 и в элементе `c[11]` – значение 78. Вы-

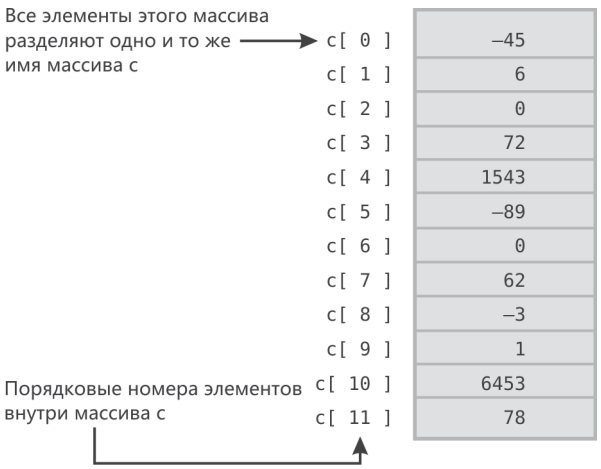


Рис. 6.1 | Массив с 12 элементами

вести сумму значений первых трех элементов массива `c` можно с помощью следующей инструкции:

```
printf( "%d", c[ 0 ] + c[ 1 ] + c[ 2 ] );
```

Разделить значение элемента `c[6]` на 2 и присвоить результат переменной `x` можно с помощью следующей инструкции:

```
x = c[ 6 ] / 2;
```

Квадратные скобки используются для обозначения индекса массива и фактически являются *оператором* языка C. Они имеют тот же уровень приоритета, что и *оператор вызова функции* (то есть круглые скобки, помещаемые после имени функции в ее вызове). В табл. 6.1 указаны приоритеты и ассоциативность операторов, с которыми мы познакомились к данному моменту.

Таблица 6.1 | Приоритеты и ассоциативность операторов

Операторы	Ассоциативность	Тип
`[] ()` ++ (постфиксный) -- (постфиксный)	Справа налево	Постфиксный
`+ -` ! (тип) `++` (префиксный) `- -` (префиксный)	Справа налево	Унарный
* / %	Слева направо	Мультипликативный
+ -	Слева направо	Аддитивный
< <= > >=	Слева направо	Проверка отношения
== !=	Слева направо	Проверка на равенство
&&	Слева направо	Логическое И
||	Слева направо	Логическое ИЛИ

Таблица 6.1 | (*окончание*)

Операторы	Ассоциативность	Тип
?:	Справа налево	Условный
= += -= *= /= %=	Справа налево	Присваивание
,	Слева направо	Запятая

6.3 Определение массивов

В определении массива необходимо указать тип элементов массива и их количество, чтобы компилятор мог зарезервировать соответствующий объем памяти. Следующее определение резервирует память для массива целых чисел из 12 элементов, которые доступны по индексам в диапазоне 0–11.

```
int c[ 12 ];
```

Определение

```
int b[ 100 ], x[ 27 ];
```

резервирует память для массива целых чисел `b`, содержащего 100 элементов, и массива целых чисел `x`, содержащего 27 элементов. Эти массивы индексируются числами в диапазоне 0–99 и 0–26 соответственно.

Массивы могут также хранить данные других типов. Например, массив типа `char` может хранить строку символов. Строки символов и их сходство с массивами будут обсуждаться в главе 8. А отношения между указателями и массивами – в главе 7.

6.4 Примеры массивов

В этом разделе будет представлено несколько примеров, демонстрирующих порядок определения и инициализации массивов, а также выполнения типичных операций с ними.

Определение массива и инициализация его элементов в цикле

Как и любые другие переменные, неинициализированные элементы массива содержат «мусор». В примере 6.1 демонстрируется применение инструкции `for` для инициализации элементов массива целых чисел `n` нулевыми значениями и вывода содержимого массива в табличной форме. Первая инструкция `printf` (строка 16) выводит заголовки столбцов, содержимое которых выводится следующей ниже инструкцией `for`.

Пример 6.1 | Инициализация элементов массива нулями

```
1 // Пример 6.1: fig06_03.c
2 // Инициализация элементов массива нулями.
3 #include <stdio.h>
4
```

```
 5 // выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    int n[ 10 ]; // n - массив из 10 целых чисел
 9    size_t i;    // счетчик
10
11    // инициализировать элементы массива n нулями
12    for ( i = 0; i < 10; ++i ) {
13       n[ i ] = 0;    // записать 0 в элемент с индексом i
14    } // конец for
15
16    printf( "%s%13s\n", "Element", "Value" );
17
18    // вывести содержимое массива n в табличной форме
19    for ( i = 0; i < 10; ++i ) {
20       printf( "%7u%13d\n", i, n[ i ] );
21    } // конец for
22 } // конец main
```

```
Element         Value
      0             0
      1             0
      2             0
      3             0
      4             0
      5             0
      6             0
      7             0
      8             0
      9             0
```

Переменная `i` объявлена с типом `size_t` (строка 9), который, согласно стандарту языка C, представляет целые числа без знака. Этот тип рекомендуется использовать для переменных, представляющих размеры или индексы массивов. Тип `size_t` объявлен в заголовочном файле `<stddef.h>`, который часто подключается в других заголовочных файлах (таких как `<stdio.h>`). [*Обратите внимание*: если при попытке скомпилировать пример 6.1 вы получите сообщение об ошибке, просто подключите заголовочный файл `<stddef.h>`.]

Инициализация массива в определении списком инициализации

Элементы массива можно также инициализировать в инструкции определения массива, добавив после определения знак «равно» (=) и перечислив через запятую начальные значения элементов в фигурных скобках {}. Пример 6.2 инициализирует массив целых чисел 10 значениями (строка 9) и выводит его в табличной форме.

Пример 6.2 | Инициализация элементов массива списком инициализации

```
1 // Пример 6.2: fig06_04.c
2 // Инициализация элементов массива списком инициализации.
3 #include <stdio.h>
```

```
 4
 5 // выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    // инициализировать массив n списком инициализации
 9    int n[ 10 ] = { 32, 27, 64, 18, 95, 14, 90, 70, 60, 37 };
10    size_t i; // счетчик
11
12    printf( "%s%13s\n", "Element", "Value" );
13
14    // вывести содержимое массива в табличной форме
15    for ( i = 0; i < 10; ++i ) {
16       printf( "%7u%13d\n", i, n[ i ] );
17    } // конец for
18 } // конец main
```

```
Element         Value
      0            32
      1            27
      2            64
      3            18
      4            95
      5            14
      6            90
      7            70
      8            60
      9            37
```

Если в списке инициализации значений окажется меньше, чем элементов в массиве, оставшиеся элементы будут инициализированы нулями. Например, элементы массива `n` из примера 6.1 можно было бы инициализировать нулями следующей инструкцией:

```
int n[ 10 ] = { 0 }; // инициализирует все элементы массива нулями
```

Эта инструкция *явно* инициализирует первый элемент нулем и неявно – остальные девять элементов, потому что при нехватке инициализирующих значений остальные элементы массива по умолчанию инициализируются нулями. Об этом следует помнить, так как элементы массивов *не* инициализируются нулевыми значениями автоматически. Необходимо инициализировать нулем хотя бы первый элемент, чтобы автоматически обнулить остальные. Инициализация элементов *статических* массивов выполняется перед запуском программы, а *автоматических* – во время выполнения.

Иногда программисты забывают инициализировать элементы массива.

Определение массива

```
int n[ 5 ] = { 32, 27, 64, 18, 95, 14 };
```

вызовет синтаксическую ошибку во время компиляции, потому что здесь определено шесть инициализирующих значений, а массив содержит *только* пять элементов.

Если количество инициализирующих значений превышает количество элементов в массиве, это вызовет синтаксическую ошибку во время компиляции.

Если размер массива не указан в объявлении, содержащем инициализирующие значения, количество элементов будет определено, исходя из количества значений в списке инициализации. Например, инструкция

```
int n[] = { 1, 2, 3, 4, 5 };
```

создаст массив с пятью элементами, инициализированными указанными значениями.

Определение размера массива с помощью символической константы и инициализация элементов массива результатами вычислений

В примере 6.3 выполняется инициализация элементов массива `s` значениями 2, 4, 6, …, 20 и осуществляется вывод содержимого этого массива в табличной форме. Значения генерируются умножением счетчика цикла на число 2 и прибавлением к произведению числа 2.

Пример 6.3 | Инициализация элементов массива s четными целыми числами от 2 до 20

```
// Пример 6.3: fig06_05.c
// Инициализация элементов массива s четными целыми числами от 2 до 20.
#include <stdio.h>
#define SIZE 10 // максимальный размер массива

// выполнение программы начинается с функции main
int main( void )
{
   // константу SIZE можно использовать для определения размера массива
   int s[ SIZE ]; // массив s имеет SIZE элементов
   size_t j;      // счетчик

   for ( j = 0; j < SIZE; ++j ) { // инициализировать элементы массива
      s[ j ] = 2 + 2 * j;
   } // конец for

   printf( "%s%13s\n", "Element", "Value" );

   // вывести содержимое массива s в табличной форме
   for ( j = 0; j < SIZE; ++j ) {
      printf( "%7u%13d\n", j, s[ j ] );
   } // конец for
} // конец main
```

```
Element      Value
      0          2
      1          4
      2          6
      3          8
      4         10
      5         12
      6         14
      7         16
      8         18
      9         20
```

В этой программе появилась новая для нас директива препроцессора `#define`. Строка 4

```
#define SIZE 10
```

определяет **символическую константу** `SIZE` со значением `10`. Символическая константа – это идентификатор, который замещается препроцессором на указанный текст перед передачей исходного кода компилятору. Когда программа будет обрабатываться препроцессором, он заменит все вхождения символической константы `SIZE` текстом `10`. Использование символических констант для определения размеров массивов делает программы более **масштабируемыми**. Цикл `for` (строка 13) в примере 6.3 легко мог бы выполнить инициализацию массива с 1000 элементов, для чего достаточно просто изменить значение константы `SIZE` в директиве `#define`, заменив `10` на `1000`. Если бы мы не использовали константу `SIZE`, нам пришлось бы выполнить замену в трех разных местах. С ростом объема программ этот прием становится еще более практичным.

Завершать директивы `#define` и `#include` препроцессора точкой с запятой, является ошибкой. Не забывайте, что директивы препроцессора не являются инструкциями языка C.

Если директиву `#define` препроцессора в строке 4 завершить точкой с запятой, препроцессор заменит все вхождения символической константы `SIZE` текстом `10;`. Это может вызвать синтаксическую ошибку на этапе компиляции или логическую – во время выполнения. Помните, что препроцессор – это ***не*** компилятор.

Определение размеров всех массивов с помощью символических констант повышает масштабируемость программ.

Попытка присвоить значение символической константе в выполняемой инструкции вызовет синтаксическую ошибку. Символическая константа – это не переменная. Компилятор не резервирует память для хранения символических констант, как для переменных, хранящих значения во время выполнения.

Для составления имен символических констант используйте только буквы верхнего регистра. Это позволит визуально отличать имена таких констант от окружающего текста программы и послужит дополнительным напоминанием, что символические константы не являются переменными.

В именах символических констант, состоящих из нескольких слов, отделяйте слова символами подчеркивания, что сделает их более удобочитаемыми.

Суммирование элементов массива

В примере 6.4 демонстрируется вычисление суммы 12 целочисленных значений элементов массива `a`. Все вычисления выполняются в теле цикла `for` (строка 16).

Пример 6.4 | Вычисление суммы значений элементов массива

```
 1 // Пример 6.4: fig06_06.c
 2 // Вычисление суммы значений элементов массива.
 3 #include <stdio.h>
 4 #define SIZE 12
 5
 6 // выполнение программы начинается с функции main
 7 int main( void )
 8 {
 9    // инициализация массива выполняется списком инициализации
10    int a[ SIZE ] = { 1, 3, 5, 4, 7, 2, 99, 16, 45, 67, 89, 45 };
11    size_t i;       // счетчик
12    int total = 0; // сумма
13
14    // вычислить сумму элементов массива a
15    for ( i = 0; i < SIZE; ++i ) {
16       total += a[ i ];
17    } // конец for
18
19    printf( "Total of array element values is %d\n", total );
20 } // конец main
```

```
Total of array element values is 383
```

Использование массивов для суммирования результатов исследований

Наш следующий пример использует массивы для суммирования результатов исследований. Взгляните на описание стоящей перед нами задачи.

> *Сорока студентам было предложено оценить качество блюд в студенческой столовой и выставить балл от 1 до 10 (1 означает ужасное качество, а 10 – превосходное). Требуется поместить эти 40 ответов в массив и выполнить оценку результатов голосования.*

Это – типичная область применения массивов (см. пример 6.5). Мы собираемся определить количество каждой оценки. Массив `responses` (стро-

ка 17) – это 40-элементный массив с ответами студентов. Для подсчета оценок мы будем использовать 11-элементный массив `frequency` (строка 14). Элемент `frequency[0]` не применяется, потому что логичнее хранить количество оценок 1 в элементе `frequency[1]`, а не в элементе `frequency[0]`. Кроме того, это дает нам возможность использовать каждую оценку непосредственно в качестве индекса в массиве `frequency`.

Пример 6.5 | Анализ результатов опроса студентов

```
 1 // Пример 6.5: fig06_07.c
 2 // Анализ результатов опроса студентов.
 3 #include <stdio.h>
 4 #define RESPONSES_SIZE 40 // размеры массивов
 5 #define FREQUENCY_SIZE 11
 6
 7 // выполнение программы начинается с функции main
 8 int main( void )
 9 {
10    size_t answer; // счетчик для цикла, выполняющего обход 40 ответов
11    size_t rating; // счетчик для цикла, выполняющего подсчет оценок 1-10
12
13    // инициализировать элементы массива frequency нулями
14    int frequency[ FREQUENCY_SIZE ] = { 0 };
15
16    // поместить результаты опроса в массив responses
17    int responses[ RESPONSES_SIZE ] = { 1, 2, 6, 4, 8, 5, 9, 7, 8, 10,
18         1, 6, 3, 8, 6, 10, 3, 8, 2, 7, 6, 5, 7, 6, 8, 6, 7, 5, 6, 6,
19         5, 6, 7, 5, 6, 4, 8, 6, 8, 10 };
20
21    // для каждого ответа получить значение элемента массива responses
22    // и использовать его в качестве индекса в массиве frequency,
23    // чтобы определить, какой элемент увеличивать
24    for ( answer = 0; answer < RESPONSES_SIZE; ++answer ) {
25       ++frequency[ responses [ answer ] ];
26    } // конец for
27
28    // вывести результаты
29    printf( "%s%17s\n", "Rating", "Frequency" );
30
31    // вывести количество каждой оценки в табличной форме
32    for ( rating = 1; rating < FREQUENCY_SIZE; ++rating ) {
33       printf( "%6d%17d\n", rating, frequency[ rating ] );
34    } // конец for
35 } // конец main
```

```
Rating        Frequency
     1                2
     2                2
     3                2
     4                2
     5                5
     6               11
     7                5
     8                7
     9                1
    10                3
```

Цикл `for` (строка 24) обрабатывает ответы по одному, извлекая их из массива `responses`, и увеличивает один из 10 счетчиков (от `frequency[1]` до `frequency[10]`) в массиве `frequency`. Основой цикла является инструкция в строке 25

```
++frequency[ responses[ answer ] ];
```

которая увеличивает соответствующий счетчик в зависимости от значения `responses[answer]`. Когда переменная счетчика `answer` получает значение 0, выражение `responses[answer]` возвращает `1`, поэтому инструкция `++frequency[responses[answer]];` интерпретируется как

```
++frequency[ 1 ];
```

которая увеличивает элемент массива `frequency` с индексом 1. Когда переменная `answer` получает значение 2, выражение `responses[answer]` возвращает `6`, поэтому инструкция `++frequency[responses[answer]];` интерпретируется как

```
++frequency[ 6 ];
```

то есть как элемент массива `frequency` с индексом 6, и т. д. Независимо от количества ответов для суммирования результатов достаточно 11-элементного (в котором элемент с индексом 0 просто игнорируется). Если в данные попадет недопустимое значение, например 13, программа попытается увеличить элемент `frequency[13]`, что приведет к выходу за границы массива. *В языке C не выполняется проверка выхода за границы массива, позволяющая предотвратить обращение к несуществующему элементу.* Поэтому программа может записать значение в не предназначенную для этого область памяти и породить тем самым проблему безопасности, о которой рассказывается в разделе 6.11. Вы должны гарантировать допустимость оценок, чтобы не выйти за границы массива.

Попытка сослаться на элемент за границами массива является ошибкой.

При выполнении обхода элементов массива индекс никогда не должен принимать отрицательные значения и всегда должен быть меньше общего количества элементов в массиве (размер – 1). Убедитесь, что условие завершения цикла не позволит обратиться к элементам за пределами массива.

Программы должны проверять допустимость всех входных значений, чтобы предотвратить нежелательное влияние ошибочных данных на результаты вычислений.

Отображение значений элементов массива в виде гистограммы

Наш следующий пример (пример 6.6) читает числа из массива и отображает полученную информацию в форме столбчатой диаграммы, или гистограммы, – сначала выводится само число, а затем строка с соответствующим количеством звездочек. Столбики в этом примере выводит вложенная инструкция `for` (строка 20). Обратите внимание на инструкцию `puts( "" )` в конце вывода каждого столбика (строка 24).

Пример 6.6 | Вывод гистограммы

```
 1 // Пример 6.6: fig06_08.c
 2 // Вывод гистограммы.
 3 #include <stdio.h>
 4 #define SIZE 10
 5
 6 // выполнение программы начинается с функции main
 7 int main( void )
 8 {
 9    // инициализировать массив n
10    int n[ SIZE ] = { 19, 3, 15, 7, 11, 9, 13, 5, 17, 1 };
11    size_t i; // счетчик для внешнего цикла for
12    int j;    // счетчик для внутреннего цикла for
13
14    printf( "%s%13s%17s\n", "Element", "Value", "Histogram" );
15
16    // для каждого элемента массива n вывести столбик гистограммы
17    for ( i = 0; i < SIZE; ++i ) {
18       printf( "%7u%13d        ", i, n[ i ]) ;
19
20       for ( j = 1; j <= n[ i ]; ++j ) { // вывести один столбик
21          printf( "%c", '*' );
22       } // конец внутреннего цикла for
23
24       puts( "" ); // конец столбика гистограммы
25    } // конец внешнего цикла for
26 } // конец main
```

```
Element       Value       Histogram
      0          19       *******************
      1           3       ***
      2          15       ***************
      3           7       *******
      4          11       ***********
      5           9       *********
      6          13       *************
      7           5       *****
      8          17       *****************
      9           1       *
```

Бросание кубика 6 000 000 раз и суммирование результатов в массиве

В главе 5 говорилось, что в этой главе будет показана более элегантная реализация программы из примера 5.5. Задача состоит в том, чтобы сымитиро-

вать 6 000 000 бросков игрового кубика и определить, действительно ли генератор случайных чисел производит случайные числа. Версия программы, основанная на применении массива, представлена в примере 6.7.

Пример 6.7 | Бросание кубика 6 000 000 раз

```
// Пример 6.7: fig06_09.c
// Бросание кубика 6 000 000 раз
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define SIZE 7

// выполнение программы начинается с функции main
int main( void )
{
   size_t face;       // случайное значение 1-6
   unsigned int roll; // счетчик бросков 1-6 000 000
   unsigned int frequency[ SIZE ] = { 0 }; // сбросить счетчики

   srand( time( NULL ) ); // инициализировать генератор случайных чисел

   // бросить кубик 6 000 000 раз
   for ( roll = 1; roll <= 6000000; ++roll ) {
      face = 1 + rand() % 6;
      ++frequency[ face ]; // замена инструкции switch в примере 5.3
   } // конец for

   printf( "%s%17s\n", "Face", "Frequency" );

   // вывести количества выпадений 1-6 в табличной форме
   for ( face = 1; face < SIZE; ++face ) {
      printf( "%4d%17d\n", face, frequency[ face ] );
   } // конец for
} // конец main
```

```
Face       Frequency
   1          999753
   2         1000773
   3          999600
   4          999786
   5         1000552
   6          999536
```

Использование массивов символов для работы со строками

До сих пор мы рассматривали только массивы целых чисел. Однако массивы способны хранить данные *любого* типа. В этом подразделе мы узнаем, как хранить *строки* в массивах символов. До настоящего момента мы сталкивались со строками только в инструкциях вывода `printf`. В действительности в языке C строки, такие как `"hello"`, являются массивами символов.

Массивы символов обладают некоторыми уникальными характеристиками. Массив символов можно инициализировать строковым литералом. Например, инструкция

```
char string1[] = "first";
```

инициализирует элементы массива `string1` отдельными символами из литерала строки `"first"`. В этом случае размер массива `string1` определяется компилятором, исходя из длины строки. Строка `"first"` содержит пять символов плюс один специальный символ, завершающий строку, – **нулевой символ**. То есть фактически массив `string1` содержит *шесть* элементов. Нулевой символ представлен константой `'\0'`. Все строки в языке C завершаются этим символом. Массив символов, представляющий строку, всегда должен иметь размер, достаточный, чтобы вместить все символы строки плюс нулевой символ в конце.

Массив символов также можно инициализировать списком отдельных символов, но это довольно утомительно. Следующее определение эквивалентно предыдущему:

```
char string1[] = { 'f', 'i', 'r', 's', 't', '\0' };
```

Так как строка в действительности является массивом символов, к отдельным символам в строке можно обращаться непосредственно, используя индексы. Например, выражение `string1[0]` соответствует символу `'f'`, а `string1[3]` – символу `'s'`.

Строки также можно вводить с клавиатуры непосредственно в массивы символов, с помощью функции `scanf` и спецификатора преобразования `%s`. Например, инструкция

```
char string2[ 20 ];
```

создаст массив, в котором можно будет сохранить строку длиной до 19 символов плюс завершающий нулевой символ, а инструкция

```
scanf( "%19s", string2 );
```

прочитает строку с клавиатуры и сохранит ее в массиве `string2`. Имя массива в данном случае передается функции `scanf` без предшествующего оператора `&`, используемого при передаче нестроковых переменных. Оператор `&` обычно применяется, чтобы передать функции `scanf` адрес переменной в памяти, куда должно быть сохранено прочитанное значение. В разделе 6.5, где рассматривается тема передачи массивов функциям, мы узнаем, что значением имени массива является *адрес начала массива*; поэтому использовать оператор `&` с массивами не требуется. Функция `scanf` будет читать символы, пока не встретит символ *пробела*, *табуляции*, *перевода строки* или *признак конца файла*. Строка для записи в массив `string2` должна быть не длиннее 19 символов, чтобы осталось место для завершающего нулевого символа. Если пользователь введет 20 или более символов, программа может завершиться аварийно или породить уязвимость в системе безопасности. По этой причине был использован спецификатор `%19s`, ограничивающий функцию

scanf 19 символами и не позволяющий записать лишние символы в память за пределами массива string2.

Вся ответственность за наличие достаточного количества элементов в массиве, куда выполняется запись введенных пользователем данных, возлагается на плечи программиста. Функция scanf не проверяет размер массива. То есть scanf может выполнить запись и за пределы массива.

Массив символов, представляющий строку, можно вывести с помощью функции printf и спецификатора %s. Следующая инструкция выведет содержимое массива string2:

```
printf( "%s\n", string2 );
```

Функция printf, как и scanf, *не* проверяет размер массива. Она продолжает выводить символ за символом, пока не встретит завершающий нулевой символ. [Подумайте, что выведет printf, если по каким-то причинам завершающий нулевой символ отсутствует в строке.]

В примере 6.8 демонстрируется инициализация массива символов строковым литералом, ввод строки в массив символов, вывод массива символов как строки и доступ к отдельным символам в строке. Программа использует инструкцию for (строка 23) для обхода в цикле элементов массива string1 и вывода отдельных символов через пробелы с помощью спецификатора %c. Условие в инструкции for остается истинным, пока значение счетчика остается меньше размера массива и *не* достигнут завершающий нулевой символ. Эта программа реализует чтение только строк без пробельных символов. Об особенностях ввода строк с пробелами будет рассказываться в главе 8. Обратите внимание, что строки 18–19 содержат два строковых литерала, разделенных пробельным символом. Компилятор автоматически объединяет такие строковые литералы в один – это очень удобно, когда приходится иметь дело с длинными литералами строк.

Пример 6.8 | Интерпретация массивов символов как строк

```
 1 // Пример 6.8: fig06_10.c
 2 // Интерпретация массивов символов как строк.
 3 #include <stdio.h>
 4 #define SIZE 20
 5
 6 // выполнение программы начинается с функции main
 7 int main( void )
 8 {
 9    char string1[ SIZE ];              // резервирует память
                                         // для 20 символов
10    char string2[] = "string literal"; // резервирует память
                                         // для 15 символов
11    size_t i;                          // счетчик
12
13    // прочитать строку, введенную пользователем в массив string1
14    printf( "%s", "Enter a string (no longer than 19 characters): " );
15    scanf( "%19s", string1 ); // ввод строки длиной до 19 символов
16
```

```
17    // вывести строки
18   printf( "string1 is: %s\nstring2 is: %s\n"
19           "string1 with spaces between characters is:\n",
20           string1, string2 );
21
22    // выводить символы до достижения нулевого символа
23    for ( i = 0; i < SIZE && string1[ i ] != '\0'; ++i ) {
24       printf( "%c ", string1[ i ] );
25    } // конец for
26
27    puts( "" );
28 } // конец main
```

```
Enter a string (no longer than 19 characters): Hello there
string1 is: Hello
string2 is: string literal
string1 with spaces between characters is:
H e l l o
```

Статические и автоматические локальные массивы

В главе 5 мы познакомились с классом хранения `static`. Статические локальные переменные продолжают существовать на *всем протяжении* выполнения программы, но *видимы* только в теле функции. Спецификатор `static` также может применяться к определениям локальных массивов, чтобы они *не* создавались и *не* инициализировались при каждом вызове функции и *не* уничтожались после выполнения функции. Это может увеличить скорость выполнения программ, особенно если они часто вызывают функции, содержащие локальные массивы большого размера.

Объявление локальных автоматических массивов статическими позволяет избежать их создания при каждом вызове, что особенно полезно в часто вызываемых функциях.

Статические массивы инициализируются один раз в момент запуска программы. Если вы не инициализируете их явно, по умолчанию статические массивы инициализируются нулями.

В примере 6.9 демонстрируется функция `staticArrayInit` (строки 21–40) с локальным статическим массивом (строка 24) и функция `automaticArrayInit` (строки 43–62) с локальным автоматическим массивом (строка 46). Функция `staticArrayInit` вызывается дважды (строки 12 и 16). Локальный статический массив (строка 24) инициализируется нулями в момент запуска программы. Функция выводит элементы массива, прибавляет число 5 к каждому элементу и выводит их еще раз. Во втором вызове функции статический массив будет содержать значения, сохранившиеся после первого вызова.

Функция `automaticArrayInit` тоже вызывается дважды (строки 13 и 17). Элементы автоматического локального массива инициализируются в функции значениями 1, 2 и 3 (строка 46). Функция выводит элементы массива, прибавляет число 5 к каждому элементу и выводит их еще раз. Во втором

вызове функции элементы функции снова будут инициализированы значениями 1, 2 и 3, потому что массив имеет автоматический класс хранения.

Будет ошибкой полагать, что элементы локального статического массива инициализируются нулями в каждом вызове функции, где он объявлен.

Пример 6.9 | Статические массивы инициализируются нулями по умолчанию, если не инициализируются явно

```
// Пример 6.9: fig06_11.c
// Статические массивы инициализируются нулями по умолчанию.
#include <stdio.h>

void staticArrayInit( void );    // прототип функции
void automaticArrayInit( void ); // прототип функции

// выполнение программы начинается с функции main
int main( void )
{
   puts( "First call to each function:" );
   staticArrayInit();
   automaticArrayInit();

   puts( "\n\nSecond call to each function:" );
   staticArrayInit();
   automaticArrayInit();
} // конец main

// функция для демонстрации локального статического массива
void staticArrayInit( void )
{
   // инициализирует элементы значением 0 при первом вызове
   static int array1[ 3 ];
   size_t i; // счетчик

   puts( "\nValues on entering staticArrayInit:" );

   // вывести содержимое array1
   for ( i = 0; i <= 2; ++i ) {
      printf( "array1[ %u ] = %d ", i, array1[ i ] );
   } // конец for

   puts( "\nValues on exiting staticArrayInit:" );

   // изменить и вывести содержимое array1
   for ( i = 0; i <= 2; ++i ) {
       printf( "array1[ %u ] = %d ", i, array1[ i ] += 5 );
   } // конец for
} // конец функции staticArrayInit

// функция для демонстрации локального автоматического массива
void automaticArrayInit( void )
{
   // инициализирует элементы при каждом вызове
   int array2[ 3 ] = { 1, 2, 3 };
   size_t i; // счетчик
```

```
48
49    puts( "\n\nValues on entering automaticArrayInit:" );
50
51    // вывести содержимое array2
52    for ( i = 0; i <= 2; ++i ) {
53       printf("array2[ %u ] = %d ", i, array2[ i ] );
54    } // конец for
55
56    puts( "\nValues on exiting automaticArrayInit:" );
57
58    // изменить и вывести содержимое array2
59    for ( i = 0; i <= 2; ++i ) {
60       printf( "array2[ %u ] = %d ", i, array2[ i ] += 5 );
61    } // конец for
62 } // конец функции automaticArrayInit
```

```
First call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 0 array1[ 1 ] = 0 array1[ 2 ] = 0
Values on exiting staticArrayInit:
array1[ 0 ] = 5 array1[ 1 ] = 5 array1[ 2 ] = 5

Values on entering automaticArrayInit:
array2[ 0 ] = 1 array2[ 1 ] = 2 array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6 array2[ 1 ] = 7 array2[ 2 ] =

Second call to each function:

Values on entering staticArrayInit:
array1[ 0 ] = 5 array1[ 1 ] = 5 array1[ 2 ] = 5
Values on exiting staticArrayInit:
array1[ 0 ] = 10 array1[ 1 ] = 10 array1[ 2 ] = 10

Values on entering automaticArrayInit:
array2[ 0 ] = 1 array2[ 1 ] = 2 array2[ 2 ] = 3
Values on exiting automaticArrayInit:
array2[ 0 ] = 6 array2[ 1 ] = 7 array2[ 2 ] = 8
```

6.5 Передача массивов функциям

Чтобы передать функции аргумент-массив, укажите в вызове имя массива без квадратных скобок. Например, если массив `hourlyTemperatures` определен как

```
int hourlyTemperatures[ HOURS_IN_A_DAY ];
```

инструкция вызова

```
modifyArray( hourlyTemperatures, HOURS_IN_A_DAY )
```

передаст массив `hourlyTemperatures` и его размер функции `modifyArray`.

Не забывайте, что в языке C все аргументы передаются *по значению*. Однако массивы в C автоматически передаются в функции *по ссылке* (в главе 7

мы убедимся, что в этом нет никаких противоречий) – вызываемая функция может изменять значения элементов оригинального массива, объявленного в вызывающем коде. Имя массива компилируется в адрес его первого элемента в памяти. Поскольку выполняется передача начального адреса массива, вызываемая функция точно знает, где хранится массив. То есть, когда вызываемая функция изменяет элементы массива, она фактически изменяет элементы *оригинального* массива.

Пример 6.10 наглядно показывает, что имя массива в действительности интерпретируется как *адрес* его первого элемента, выводя `array`, `&array[0]` и `&array` с использованием спецификатора `%p`, специально предназначенного для вывода адресов. Обычно спецификатор `%p` выводит адрес в шестнадцатеричном представлении, но это его поведение зависит от компилятора. Шестнадцатеричные числа (в системе счисления с основанием 16) состоят из цифр 0–9 и букв A–F (эти буквы являются шестнадцатеричными эквивалентами десятичных чисел 10–15). В приложении C дается более глубокое обсуждение взаимоотношений между двоичной (по основанию 2), восьмеричной (по основании 8), десятичной (по основанию 10; стандартные целые числа) и шестнадцатеричной (по основанию 16) системами счисления. Из вывода программы видно, что `array`, `&array` и `&array[0]` имеют одно и то же значение, а именно `0012FF78`. Вывод этой программы может отличаться в разных системах, но адреса всегда остаются идентичными в конкретном сеансе выполнения программы на конкретном компьютере.

Передача массивов по ссылке положительно сказывается на производительности. Если бы массивы передавались по значению, пришлось бы копировать каждый элемент массива при передаче. В случае передачи огромных массивов часто вызываемым функциям копирование их элементов могло бы приводить к существенным затратам времени.

Пример 6.10 | Имя массива интерпретируется как адрес его первого элемента

```
1 // Пример 6.10: fig06_12.c
2 // Имя массива интерпретируется как адрес его первого элемента.
3 #include <stdio.h>
4
5 // выполнение программы начинается с функции main
6 int main( void )
7 {
8    char array[ 5 ]; // определение массива с 5 элементами
9
10   printf( "    array = %p\n&array[0] = %p\n    &array = %p\n",
11      array, &array[ 0 ], &array );
12 } // конец main
```

```
    array = 0012FF78
&array[0] = 0012FF78
   &array = 0012FF78
```

Существует возможность передавать массивы по значению (используя простой трюк, описанный в главе 10).

Хотя при передаче массива целиком он передается по ссылке, при передаче отдельных элементов они передаются *по значению*, как простые переменные. Такие простые единичные элементы данных (как значения типа `int`, `float` и `char`) называются **скалярами**. Чтобы передать элемент массива функции, используется имя массива с индексом. В главе 7 будет показано, как можно передавать функциям скаляры (то есть отдельные переменные и элементы массивов) по ссылке.

В определении функции, принимающей массив, в списке параметров *должно* быть указано, что принимается именно массив. Например, заголовок функции `modifyArray` (вызывавшейся выше в этом разделе) можно записать так:

```
void modifyArray( int b[], int size )
```

указав тем самым, что `modifyArray` ожидает получить массив целых чисел в параметре `b` и количество элементов в массиве – в параметре `size`. В заголовке функции *не* требуется объявлять *размер* массива внутри квадратных скобок. Но если он указан, компилятор убедится, что он больше нуля, и затем просто будет игнорировать его. Отрицательный размер вызовет ошибку во время компиляции. Так как массивы автоматически передаются по ссылке, имя `b` в вызываемой функции фактически будет ссылаться на массив в вызывающем коде (каковым в предыдущем вызове является массив `hourlyTemperatures`). В главе 7 будут представлены другие формы записи, указывающие, что функция принимает массив. Как мы увидим, все эти формы записи опираются на родство между указателями и массивами в языке C.

Отличие передачи массива целиком от передачи отдельного элемента

Пример 6.11 демонстрирует отличие передачи массива целиком от передачи отдельного его элемента. Программа сначала выводит пять элементов массива целых чисел (строки 20–22). Затем массив `a` и его размер передаются функции `modifyArray` (строка 27), где каждый элемент массива умножается на 2 (строки 53–55). Потом в функции `main` повторяется вывод содержимого массива `a` (строки 32–34). Как видно из вывода программы, элементы массива действительно изменяются функцией `modifyArray`. После этого программа выводит значение элемента `a[3]` (строка 38) и передает его функции `modifyElement` (строка 40), которая умножает свой аргумент на 2 (строка 63) и выводит новое значение. Далее элемент `a[3]` повторно выводится в функции `main` (строка 43), откуда видно, что отдельные элементы массивов передаются функциям по значению.

В программах могут возникать ситуации, когда функции *не* должны иметь возможности изменять элементы массивов. В языке C для этих целей имеется квалификатор типа `const` (от англ. «constant» – константа), который можно использовать для предотвращения изменения массива в функции. Когда параметру-массиву в объявлении функции предшествует квалификатор `const`, элементы массива гарантированно не будут изменяться в теле функции, а любая попытка изменить какой-нибудь элемент приведет к ошибке компиляции. Это дает возможность скорректировать программу так, чтобы она даже не пыталась изменять элементы массивов.

Пример 6.11 | Передача массивов и отдельных элементов массивов функциям

```
// Пример 6.11: fig06_13.c
// Передача массивов и отдельных элементов массивов функциям.
#include <stdio.h>
#define SIZE 5

// прототипы функций
void modifyArray( int b[], size_t size );
void modifyElement( int e );

// выполнение программы начинается с функции main
int main( void )
{
   int a[ SIZE ] = { 0, 1, 2, 3, 4 }; // инициализация массива a
   size_t i;                          // счетчик

   puts( "Effects of passing entire array by reference:\n\nThe "
         "values of the original array are:" );

   // вывести оригинальный массив
   for ( i = 0; i < SIZE; ++i ) {
      printf( "%3d", a[ i ] );
   } // конец for

   puts( "" );

   // передать по ссылке массив a функции modifyArray
   modifyArray( a, SIZE );

   puts( "The values of the modified array are:" );

   // вывести содержимое измененного массива
   for ( i = 0; i < SIZE; ++i ) {
      printf( "%3d", a[ i ] );
   } // конец for

   // вывести значение a[ 3 ]
   printf( "\n\n\nEffects of passing array element "
      "by value:\n\nThe value of a[3] is %d\n", a[ 3 ] );

   modifyElement( a[ 3 ] ); // передать по значению элемент a[ 3 ]

   // вывести значение a[ 3 ]
```

```
43    printf( "The value of a[ 3 ] is %d\n", a[ 3 ] );
44 } конец main
45
46 // в функции modifyArray, "b" ссылается на оригинальный массив "a"
47 // в памяти
48 void modifyArray( int b[], size_t size )
49 {
50    size_t j; // счетчик
51
52    // умножить каждый элемент массива на 2
53    for ( j = 0; j < size; ++j ) {
54       b[ j ] *= 2; // фактически изменит оригинальный массив
55    } // конец for
56 } // конец функции modifyArray
57
58 // в функции modifyElement, "e" - это локальная копия элемента массива
59 // a[ 3 ], переданного из функции main
60 void modifyElement( int e )
61 {
62    // умножить параметр на 2
63    printf( "Value in modifyElement is %d\n", e *= 2 );
64 } // конец функции modifyElement
```

```
Effects of passing entire array by reference:

The values of the original array are:
  0  1  2  3  4
The values of the modified array are:
  0  2  4  6  8

Effects of passing array element by value:

The value of a[3] is 6
Value in modifyElement is 12
The value of a[ 3 ] is 6
```

Использование квалификатора const с параметрами-массивами

Пример 6.12 демонстрирует действие квалификатора типа `const`. Функция `tryToModifyArray` (строка 19) определяет параметр `const int b[]`. Согласно этому определению, массив `b` является *константой* и *не может* быть изменен. Вывод, следующий ниже, показывает, что во время компиляции возникла ошибка – конкретный текст сообщения об ошибке зависит от компилятора. Каждая из трех попыток изменить элементы массива в этой функции вызывает сообщение об ошибке «l-value specifies a const object» (в качестве левостороннего значения указан объект с квалификатором `const`). В главе 7 будет также рассматриваться применение квалификатора типа `const` в других контекстах.

Пример 6.12 | Использование квалификатора const с массивами

```
1 // Пример 6.12: fig06_14.c
2 // Использование квалификатора const с массивами.
3 #include <stdio.h>
4
```

```
 5 void tryToModifyArray( const int b[] ); // прототип функции
 6
 7 // выполнение программы начинается с функции main
 8 int main( void )
 9 {
10    int a[] = { 10, 20, 30 }; // инициализация массива a
11
12    tryToModifyArray( a );
13
14    printf("%d %d %d\n", a[ 0 ], a[ 1 ], a[ 2 ] );
15 } // конец main
16
17 // в функции tryToModifyArray массив b объявлен как const, поэтому его
18 // нельзя использовать для изменения оригинального массива в main.
19 void tryToModifyArray( const int b[] )
20 {
21    b[ 0 ] /= 2; // ошибка
22    b[ 1 ] /= 2; // ошибка
23    b[ 2 ] /= 2; // ошибка
24 } // конец функции tryToModifyArray
```

```
fig06_14.c(21) : error C2166: l-value specifies const object
fig06_14.c(22) : error C2166: l-value specifies const object
fig06_14.c(23) : error C2166: l-value specifies const object
```

Квалификатор типа `const` *можно применять к параметрам-массивам в определениях функций, чтобы предотвратить изменение оригинального массива в теле функции. Это еще один пример использования принципа наименьших привилегий. Функция не должна иметь возможности изменять переданный ей массив, если это действительно необходимо.*

6.6 Сортировка массивов

Сортировка данных (то есть расположение данных в определенном порядке, например по возрастанию или убыванию) является одной из важнейших задач. Банк сортирует все операции по номерам счетов, чтобы в конце месяца подготовить выписку о состоянии счета для каждого из них. Телефонные компании сортируют списки своих клиентов по фамилии и имени, чтобы упростить поиск их телефонных номеров. Практически во всех организациях требуется сортировать некоторые данные, и в большинстве случаев объемы этих данных огромны. Сортировка данных является интереснейшей задачей, на исследование различных путей решения которой было затрачено больше всего усилий программистов. В этой главе мы обсудим самый простой способ сортировки, а в главе 12 и в приложении D познакомимся с более сложными способами, обладающими высшей производительностью.

Нередко самый простой алгоритм обладает невысокой производительностью. Основное достоинство таких алгоритмов – простота в реализации, тестировании и отладке. Чтобы добиться максимальной производительности, часто приходится применять более сложные алгоритмы.

Пример 6.13 сортирует элементы массива `a` (строка 10) в порядке возрастания. Алгоритм сортировки, реализованный здесь, называется **пузырьковой сортировкой**, потому что наименьшие значения постепенно «всплывают» вверх к началу массива, подобно пузырькам воздуха в воде, а большие, наоборот, «тонут», опускаясь на «дно» массива. Суть алгоритма состоит в том, чтобы выполнить несколько итераций по элементам массива. В каждой итерации сравниваются значения двух соседних элементов (элемента 0 и элемента 1, затем элемента 1 и элемента 2 и т. д.). Если значение элемента с меньшим индексом меньше или равно значению элемента с меньшим индексом, значения элементов оставляются как есть. В противном случае значения элементов меняются местами.

Пример 6.13 | Сортировка элементов массива в порядке возрастания их значений

```
 1 // Пример 6.13: fig06_15.c
 2 // Сортировка элементов массива в порядке возрастания их значений.
 3 #include <stdio.h>
 4 #define SIZE 10
 5
 6 // выполнение программы начинается с функции main
 7 int main( void )
 8 {
 9    // инициализировать массив a
10    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
11    int pass; // счетчик проходов
12    size_t i; // счетчик сравнений
13    int hold; // временное хранилище для обмена элементов
14
15    puts( "Data items in original order" );
16
17    // вывести оригинальный массив
18    for ( i = 0; i < SIZE; ++i ) {
19       printf( "%4d", a[ i ] );
20    } // конец for
21
22    // пузырьковая сортировка
23    // цикл, управляющий проходами
24    for ( pass = 1; pass < SIZE; ++pass ) {
25
26       // цикл, управляющий количеством сравнений в одном проходе
27       for ( i = 0; i < SIZE - 1; ++i ) {
28
29          // сравнить соседние элементы и поменять их местами,
30          // если первый элемент больше второго
31          if ( a[ i ] > a[ i + 1 ] ) {
32             hold = a[ i ];
33             a[ i ] = a[ i + 1 ];
34             a[ i + 1 ] = hold;
35          } // конец if
36       } // конец внутреннего for
37    } // конец внешнего for
38
39    puts( "\nData items in ascending order" );
40
41    // вывести отсортированный массив
```

```
42    for ( i = 0; i < SIZE; ++i ) {
43       printf( "%4d", a[ i ] );
44    } // конец for
45
46    puts( "" );
47 } // конец main
```

```
Data items in original order
2 6 4 8 10 12 89 68 45 37
Data items in ascending order
2 4 6 8 10 12 37 45 68 89
```

Сначала программа сравнивает элементы `a[0]` и `a[1]`, затем `a[1]` и `a[2]`, потом `a[2]` и `a[3]` и т. д., пока проход не завершится сравнением `a[8]` и `a[9]`. Для массива из 10 элементов выполняется только 9 сравнений. Из-за последовательного сравнения элементов наибольшее значение может переместиться вправо в массиве на много позиций за один проход, но маленькое значение переместится влево только на одну позицию.

После первого прохода наибольшее значение гарантированно окажется в последнем элементе массива `a[9]`. После второго прохода второе по величине значение гарантированно окажется в предпоследнем элементе `a[8]`. После девятого прохода девятое по величине значение гарантированно окажется в элементе `a[1]`. В результате самое маленькое значение гарантированно окажется в элементе `a[0]` также после девятого прохода. То есть для сортировки данного массива достаточно выполнить *девять* проходов, даже при том, что в нем содержится *десять* элементов.

Сортировка выполняется вложенными друг в друга циклами `for` (строки 24–37). При необходимости обмен элементов значениями выполняется тремя инструкциями присваивания:

```
32             hold = a[ i ];
33             a[ i ] = a[ i + 1 ];
34             a[ i + 1 ] = hold;
```

где дополнительная переменная `hold` служит для *временного* хранения одного из значений. Обмен нельзя выполнить двумя инструкциями присваивания:

```
a[ i ] = a[ i + 1 ];
a[ i + 1 ] = a[ i ];
```

Если, например, `a[i]` имеет значение 7, а `a[i + 1]` – значение 5, после первой инструкции присваивания оба элемента будут хранить значение 5, а значение 7 будет потеряно, именно поэтому необходима дополнительная переменная `hold`.

Главное достоинство алгоритма пузырьковой сортировки – простота реализации. Однако это медленный алгоритм, потому что на каждом проходе легчайшие элементы перемещаются только на одну позицию влево. Медлительность алгоритма становится особенно очевидной при сортировке достаточно больших массивов. Со временем были придуманы другие алгоритмы сортировки, более эффективные, чем алгоритм пузырьковой сортировки.

6.7 Пример: вычисление математического ожидания, медианы и моды

Теперь рассмотрим более крупный пример. Компьютеры часто используются для анализа результатов голосований и опросов. Программа в примере 6.14 использует массив `response`, инициализированный ответами 99 человек. Каждый ответ – это число от 1 до 9. Программа вычисляет математическое ожидание, медиану и моду 99 значений. В примере 6.15 демонстрируются результаты выполнения этой программы. Данный пример включает реализацию основных операций с массивами, необходимых при решении задач на основе массивов, включая передачу массивов функциям.

Пример 6.14 | Анализ результатов опроса: вычисление математического ожидания, медианы и моды

```
// Пример 6.14: fig06_16.c
// Анализ результатов опроса:
// вычисление математического ожидания, медианы и моды.
#include <stdio.h>
#define SIZE 99

// прототипы функций
void mean( const unsigned int answer[] );
void median( unsigned int answer[] );
void mode( unsigned int freq[], const unsigned int answer[] ) ;
void bubbleSort( int a[] );
void printArray( const unsigned int a[] );

// выполнение программы начинается с функции main
int main( void )
{
   unsigned int frequency[ 10 ] = { 0 }; // инициализировать массив
                                         // frequency

   // инициализировать массив response
   unsigned int response[ SIZE ] =
      { 6, 7, 8, 9, 8, 7, 8, 9, 8, 9,
        7, 8, 9, 5, 9, 8, 7, 8, 7, 8,
        6, 7, 8, 9, 3, 9, 8, 7, 8, 7,
        7, 8, 9, 8, 9, 8, 9, 7, 8, 9,
        6, 7, 8, 7, 8, 7, 9, 8, 9, 2,
        7, 8, 9, 8, 9, 8, 9, 7, 5, 3,
        5, 6, 7, 2, 5, 3, 9, 4, 6, 4,
        7, 8, 9, 6, 8, 7, 8, 9, 7, 8,
        7, 4, 4, 2, 5, 3, 8, 7, 5, 6,
        4, 5, 6, 1, 6, 5, 7, 8, 7 };

   // обработать ответы
   mean( response );
   median( response );
   mode( frequency, response );
} // конец main

// вычисляет среднее арифметическое для всех ответов
void mean( const unsigned int answer[] )
```

```
40 {
41    size_t j;               // счетчик для суммирования элементов
42    unsigned int total = 0; // переменная для хранения суммы
43
44    printf( "%s\n%s\n%s\n", "********", " Mean", "********" );
45
46    // вычислить сумму ответов
47    for ( j = 0; j < SIZE; ++j ) {
48       total += answer[ j ];
49    } // конец for
50
51    printf( "The mean is the average value of the data\n"
52            "items. The mean is equal to the total of\n"
53            "all the data items divided by the number\n"
54            "of data items ( %u ). The mean value for\n"
55            "this run is: %u / %u = %.4f\n\n",
56            SIZE, total, SIZE, ( double ) total / SIZE );
57 } // конец функции mean
58
59 // сортирует массив и определяет значение медианы (срединного элемента)
60 void median( unsigned int answer[] )
61 {
62    printf( "\n%s\n%s\n%s\n%s",
63            "********", " Median", "********",
64            "The unsorted array of responses is" );
65
66    printArray( answer ); // вывести неотсортированный массив
67
68    bubbleSort( answer ); // отсортировать массив
69
70    printf( "%s", "\n\nThe sorted array is" );
71    printArray( answer ); // вывести отсортированный массив
72
73    // вывести значение медианы (срединного элемента)
74    printf( "\n\nThe median is element %u of\n"
75            "the sorted %u element array.\n"
76            "For this run the median is %u\n\n",
77            SIZE / 2, SIZE, answer[ SIZE / 2 ] );
78 } // конец функции median
79
80 // определяет модальный (наиболее часто встречающийся) ответ
81 void mode( unsigned int freq[], const unsigned int answer[] )
82 {
83    size_t rating;  // счетчик для доступа к элементам массива freq
84    size_t j;       // счетчик для доступа к элементам массива answer
85    unsigned int h; // счетчик для вывода гистограмм
86    unsigned int largest = 0;   // наибольшая частота встречаемости
87    unsigned int modeValue = 0; // наиболее часто встречаемый ответ
88
89    printf( "\n%s\n%s\n%s\n",
90            "********", " Mode", "********" );
91
92    // инициализировать массив частот нулями
93    for ( rating = 1; rating <= 9; ++rating ) {
94       freq[ rating ] = 0;
95    } // конец for
96
97    // подсчитать частоты
98    for ( j = 0; j < SIZE; ++j ) {
99       ++freq[ answer[ j ] ];
100   } // конец for
```

```
101
102    // вывести заголовки столбцов таблицы с результатами
103    printf( "%s%11s%19s\n\n%54s\n%54s\n\n",
104            "Response", "Frequency", "Histogram",
105            "1   1   2   2", "5   0   5   0   5");
106
107    // вывести результаты
108    for ( rating = 1; rating <= 9; ++rating ) {
109       printf( "%8u%11u ", rating, freq[ rating ] );
110
111       // отслеживать модальное значение и наибольшую частоту
112       if ( freq[ rating ] > largest ) {
113          largest = freq[ rating ];
114          modeValue = rating;
115       } // конец if
116
117       // вывести столбец гистограммы для данной частоты
118       for ( h = 1; h <= freq[ rating ]; ++h ) {
119          printf( "%s", "*" );
120       } // конец внутреннего цикла for
121
122       puts( "" ); // перевод строки
123    } // конец внешнего цикла for
124
125    // вывести модальное значение
126    printf( "\nThe mode is the most frequent value.\n"
127            "For this run the mode is %u which occurred"
128            " %u times.\n", modeValue, largest );
129 } // конец функции mode
130
131 // функция сортировки масссива по алгоритму пузырьковой сортировки
132 void bubbleSort( unsigned int a[] )
133 {
134    unsigned int pass; // счетчик проходов
135    size_t j;          // счетчик сравнений
136    unsigned int hold; // временное хранилище для обмена элементов
137
138    // цикл, управляющий проходами
139    for ( pass = 1; pass < SIZE; ++pass ) {
140
141       // цикл, управляющий количеством сравнений в одном проходе
142       for ( j = 0; j < SIZE - 1; ++j ) {
143
144          // поменять элементы, если необходимо
145          if ( a[ j ] > a[ j + 1 ] ) {
146             hold = a[ j ];
147             a[ j ] = a[ j + 1 ];
148             a[ j + 1 ] = hold;
149          } // конец if
150       } // конец внутреннего цикла for
151    } // конец внешнего цикла for
152 } // конец функции bubbleSort
153
154 // выводит содержимое массива (по 20 значений в строке)
155 void printArray( const unsigned int a[] )
156 {
157    size_t j; // счетчик
158
159    // вывести содержимое массива
160    for ( j = 0; j < SIZE; ++j ) {
161
```

```
162       if ( j % 20 == 0 ) { // переносить строку через каждые 20 значений
163          puts( "" );
164       } // конец if
165
166       printf( "%2u", a[ j ] );
167    } // конец for
168 } // конец функции printArray
```

Пример 6.15 | Результаты выполнения программы анализа данных

```
********
Mean
********
The mean is the average value of the data
items. The mean is equal to the total of
all the data items divided by the number
of data items ( 99 ). The mean value for
this run is: 681 / 99 = 6.8788


********
Median
********
The unsorted array of responses is
 6 7 8 9 8 7 8 9 8 9 7 8 9 5 9 8 7 8 7 8
 6 7 8 9 3 9 8 7 8 7 7 8 9 8 9 8 9 7 8 9
 6 7 8 7 8 7 9 8 9 2 7 8 9 8 9 8 9 7 5 3
 5 6 7 2 5 3 9 4 6 4 7 8 9 6 8 7 8 9 7 8
 7 4 4 2 5 3 8 7 5 6 4 5 6 1 6 5 7 8 7

The sorted array is
 1 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 5 5 5
 5 6 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7 7
 7 7 7 7 7 7 7 7 7 7 7 7 7 8 8 8 8 8 8
 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

The median is element 49 of
the sorted 99 element array.
For this run the median is 7

********
Mode
********
Response  Frequency         Histogram
                                  1   1   2   2
                              5   0   5   0   5
       1          1         *
       2          3         ***
       3          4         ****
       4          5         *****
       5          8         ********
       6          9         *********
       7         23         ***********************
       8         27         ***************************
       9         19         *******************

The mode is the most frequent value.
For this run the mode is 8 which occurred 27 times.
```

Математическое ожидание

Математическое ожидание (mean) – это среднее арифметическое 99 значений. Функция `mean` (пример 6.14, строки 39–57) вычисляет математическое ожидание путем деления суммы всех 99 элементов на их количество 99.

Медиана

Медиана – это *срединное* значение. Функция `median` (строки 60–78) сначала вызывает функцию `bubbleSort` (строки 132–152), чтобы отсортировать массив ответов, а затем выбирает `answer[SIZE / 2]` (срединный элемент) из отсортированного массива. Если количество элементов четное, медиана вычисляется как середина между двумя срединными элементами. Текущая реализация функции `median` не поддерживает этого случая. Для вывода массива `response` вызывается функция `printArray` (строки 155–168).

Мода

Мода – это *значение, чаще всего встречающееся в ответах*. Функция `mode` (строки 81–129) определяет модальное значение, подсчитывая количество каждого из ответов и затем выбирая ответ с наибольшим значением счетчика. Данная версия функции `mode` не обрабатывает модальные интервалы. Дополнительно функция `mode` выводит гистограмму с целью помочь графически определить модальное значение.

6.8 Поиск в массивах

Вам часто придется работать с большими объемами данных, хранящимися в массивах. Иногда бывает необходимо выявлять присутствие в массиве значения, соответствующего некоторому **ключевому значению**. Процесс нахождения определенного элемента в массиве называется **поиском**. В этом разделе мы рассмотрим два приема реализации поиска – простой **линейный поиск** и более эффективный (но более сложный) **поиск методом половинного деления**.

Линейный поиск в массиве

Алгоритм линейного поиска (реализован в примере 6.16) предусматривает сопоставление каждого элемента массива с **ключом поиска**. Так как значения элементов массива необязательно следуют в каком-то определенном порядке, искомое значение с равной вероятностью может находиться как в первом, так и в последнем элементе. То есть в среднем программе придется сравнить ключ поиска с *половиной* элементов.

Пример 6.16 | Линейный поиск в массиве

```
1 // Пример 6.16: fig06_18.c
2 // Линейный поиск в массиве.
3 #include <stdio.h>
4 #define SIZE 100
```

```
 5
 6 // прототип функции
 7 size_t linearSearch( const int array[], int key, size_t size );
 8
 9 // выполнение программы начинается с функции main
10 int main( void )
11 {
12    int a[ SIZE ];  // создать массив a
13    size_t x;       // счетчик для инициализации элементов массива a
14    int searchKey;  // искомое значение
15    size_t element; // индекс элемента с искомым значением или -1
16
17    // создать некоторые данные
18    for ( x = 0; x < SIZE; ++x ) {
19       a[ x ] = 2 * x;
20    } // конец for
21
22    puts( "Enter integer search key:" );
23    scanf( "%d", &searchKey );
24
25    // выполнить поиск значения searchKey в массиве a
26    element = linearSearch( a, searchKey, SIZE );
27
28    // вывести результат
29    if ( element != -1 ) {
30       printf( "Found value in element %d\n", element );
31    } // конец if
32    else {
33       puts( "Value not found" );
34    } // конец else
35 } // конец main
36
37 // сравнивает значение каждого элемента массива с ключом, пока не будет
38 // найдено совпадение или не встретится конец массива; возвращает индекс
39 // элемента или -1, если совпадение не найдено
40 size_t linearSearch( const int array[], int key, size_t size )
41 {
42    size_t n; // счетчик
43
44    // выполнить обход элементов массива
45    for ( n = 0; n < size; ++n ) {
46
47       if ( array[ n ] == key ) {
48          return n; // вернуть индекс найденного элемента
49       } // конец if
50    } // конец for
51
52    return -1; // ключ не найден
53 } // конец функции linearSearch
```

```
Enter integer search key:
36
Found value in element 18
```

```
Enter integer search key:
37
Value not found
```

Поиск в массиве методом половинного деления

Алгоритм линейного поиска отлично подходит для *небольших* или *несортированных* массивов. Однако для больших массивов линейный поиск оказывается *неэффективным*. Если массив отсортирован, для поиска в нем можно использовать высокоскоростной алгоритм поиска методом *половинного деления* (или *дихотомии*).

Алгоритм поиска методом половинного деления в отсортированном массиве исключает из рассмотрения *половину* из оставшихся элементов после каждого сравнения. Суть его состоит в том, чтобы выбрать элемент из *середины* и сравнить его значение с ключом поиска. Если они равны, искомое значение считается найденным и вызывающей программе возвращается индекс этого элемента. Если не равны, область поиска сокращается *вдвое*. Если искомое значение меньше значения элемента в середине массива, поиск продолжается *в первой его половине*, в противном случае – *во второй*. Если во второй попытке искомое значение не будет найдено, алгоритм продолжит поиск в одной четвертой доли оригинального массива. Поиск продолжается, пока ключ поиска не совпадет со значением среднего элемента оставшегося фрагмента массива, или пока не останется фрагмент массива, состоящий из единственного элемента, не равного ключу поиска (то есть поиск не увенчается успехом).

В худшем случае для поиска значения в массиве из 1023 элементов алгоритму поиска методом половинного деления потребуется выполнить *всего* 10 сравнений. Последовательно деля 1024 на 2, мы будем получать значения 512, 256, 128, 64, 32, 16, 8, 4, 2 и 1. Число 1024 (2^{10}) делится на 2 только 10 раз, и в конечном итоге остается число 1. Деление на 2 эквивалентно одному сравнению в алгоритме поиска методом половинного деления. Чтобы выполнить поиск в массиве из 1 048 576 (2^{20}) элементов, потребуется *не более* 20 сравнений. Чтобы выполнить поиск в массиве из миллиарда элементов, потребуется *не более* 30 сравнений. Это гигантское увеличение производительности в сравнении с алгоритмом линейного поиска, которому в среднем требуется сравнить ключ поиска с половиной элементов массива. Чтобы выполнить поиск в массиве из одного миллиарда элементов методом линейного поиска, потребуется произвести 500 миллионов сравнений, а методом дихотомии – не более 30! Максимальное количество сравнений можно определить как степень двойки, большую или равную количеству элементов в массиве.

В примере 6.17 приводится *итеративная* версия функции `binarySearch` (строки 42–72). Функция принимает четыре аргумента – массив целых чисел `b`, где выполняется поиск, искомое значение `searchKey`, нижний индекс `low` в массиве и верхний индекс `high` в массиве (последние два аргумента определяют фрагмент массива, где выполняется поиск). Если искомое значение не совпадает с элементом в середине, производится изменение нижнего `low` или верхнего `high` индекса так, чтобы ограничить область поиска. Если искомое значение *меньше* значения элемента в середине, верхний индекс `high`

устанавливается равным `middle - 1` и поиск продолжается среди элементов с индексами от `low` до `middle - 1`. Если искомое значение *больше* значения элемента в середине, нижний индекс `low` устанавливается равным `middle + 1` и поиск продолжается среди элементов с индексами от `middle + 1` до `high`. Программа выполняет поиск в массиве из 15 элементов. Первая степень 2, больше чем количество элементов, равна 16 (2^4), потому, чтобы произвести поиск в этом массиве, потребуется выполнить не более 4 сравнений. Программа использует функцию `printHeader` (строки 75–94) для вывода индексов в массиве и функцию `printRow` (строки 98–118) для вывода каждого фрагмента массива, оставшегося для поиска. Элемент в середине в каждом фрагменте выделяется звездочкой (*), чтобы показать, с каким значением сравнивается ключ поиска.

Пример 6.17 | Поиск методом половинного деления в отсортированном массиве

```
1 // Пример 6.17: fig06_19.c
2 // Поиск методом половинного деления в отсортированном массиве.
3 #include <stdio.h>
4 #define SIZE 15
5
6 // прототипы функций
7 size_t binarySearch(const int b[], int searchKey, size_t low, size_t high);
8 void printHeader( void );
9 void printRow( const int b[], size_t low, size_t mid, size_t high );
10
11 // выполнение программы начинается с функции main
12 int main( void )
13 {
14    int a[ SIZE ]; // создать массив a
15    size_t i;      // счетчик для инициализации элементов массива a
16    int key;       // искомое значение
17    size_t result; // индекс элемента с искомым значением или -1
18
19    // создать данные
20    for ( i = 0; i < SIZE; ++i ) {
21       a[ i ] = 2 * i;
22    } // конец for
23
24    printf( "%s", "Enter a number between 0 and 28: " );
25    scanf( "%d", &key );
26
27    printHeader();
28
29    // найти значение в массиве a
30    result = binarySearch( a, key, 0, SIZE - 1 );
31
32    // вывести результаты
33    if ( result != -1 ) {
34       printf( "\n%d found in array element %d\n", key, result );
35    } // конец if
36    else {
37       printf( "\n%d not found\n", key );
38    } // конец else
```

```
} // конец main

// выполняет поиск в массиве методом дихотомии
size_t binarySearch(const int b[], int searchKey, size_t low, size_t high)
{
   int middle; // значение элемента в середине

   // выполнять итерации, пока нижний индекс не станет больше верхнего
   while ( low <= high ) {

      // определить индекс элементов в середине оставшегося фрагмента
      middle = ( low + high ) / 2;

      // вывести фрагмент массива, где выполняется поиск в текущей
      // итерации
      printRow( b, low, middle, high );

      // если значение searchKey равно элементу middle, вернуть middle
      if ( searchKey == b[ middle ] ) {
         return middle;
      } // конец if

      // если searchKey меньше элемента middle, установить верхний
      // индекс
      else if ( searchKey < b[ middle ] ) {
         high = middle - 1; // продолжить поиск в нижней половине
                            // массива
      } // конец else if

      // если searchKey больше элемента middle, установить нижний индекс
      else {
         low = middle + 1; // продолжить поиск в верхней половине
                           // массива
      } // конец else
   } // конец while

   return -1; // искомое значение не найдено
} // конец функции binarySearch

// выводит заголовок
void printHeader( void )
{
   unsigned int i; // счетчик

   puts( "\nSubscripts:" );

   // вывести заголовки столбцов
   for ( i = 0; i < SIZE; ++i ) {
      printf( "%3u ", i );
   } // конец for

   puts( "" ); // перейти на новую строку

   // вывести строку из дефисов
   for ( i = 1; i <= 4 * SIZE; ++i ) {
      printf( "%s", "-" );
   } // конец for

   puts( "" ); // перейти на новую строку
```

```
 94 } // конец функции printHeader
 95
 96 // выводит одну строку с текущей частью массива,
 97 // подлежащей обработке.
 98 void printRow( const int b[], size_t low, size_t mid, size_t high )
 99 {
100    size_t i; // счетчик для обхода элементов массива b
101
102    // выполнить обход всего массива
103    for ( i = 0; i < SIZE; ++i ) {
104
105       // вывести пробелы вместо элементов за границами фрагмента
106       if ( i < low || i > high ) {
107          printf( "%s", "    ");
108       } // конец if
109       else if ( i == mid ) {       // вывести элемент в середине
110          printf( "%3d*", b[ i ] ); // отметить звездочкой
111       } // конец else if
112       else {                       // вывести остальные элементы
113          printf( "%3d ", b[ i ] );
114       } // конец else
115    } // конец for
116
117    puts( "" ); // перейти на новую строку
118 } // конец функции printRow
```

```
Enter a number between 0 and 28: 25

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
                                 16  18  20  22* 24  26  28
                                                 24  26* 28
                                                 24*

25 not found
```

```
Enter a number between 0 and 28: 8

Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12
                  8  10* 12
                  8*

8 found in array element 4
```

```
Enter a number between 0 and 28: 6
Subscripts:
  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14
-------------------------------------------------------------
  0   2   4   6   8  10  12  14* 16  18  20  22  24  26  28
  0   2   4   6*  8  10  12

6 found in array element 3
```

6.9 Многомерные массивы

Массивы в языке C могут иметь несколько индексов. Типичным примером использования массивов с несколькими индексами, которые в стандарте языка C называются **многомерными массивами**, является представление **таблиц** значений, расположенных по *строкам* и *столбцам*. Для идентификации элемента таблицы требуется указать два индекса: *первый* (по соглашениям) определяет *строку* элемента, а *второй* (по соглашениям) – *столбец*. Таблицы, или массивы, требующие указывать два индекса для идентификации конкретного элемента, называются **двумерными массивами**. Многомерные массивы могут иметь больше двух измерений.

На рис. 6.20 изображен двумерный массив `a`. Массив содержит три строки и четыре столбца, и такой массив принято называть массивом «3 на 4». В общем случае массивы с *m* строками и *n* столбцами называют массивами *m* на *n*.

Каждый элемент в массиве `a` на рис. 6.2 идентифицируется именем в форме `a[i][j]`, где `a` – это имя массива, а `i` и `j` – индексы, уникально идентифицирующие каждый элемент в `a`. Имена всех элементов в строке 0 включают первый индекс со значением `0`; имена всех элементов в столбце 3 включают второй индекс со значением `3`.

Ссылка на элемент двумерного массива как `[ x, y ]` вместо `a[ x ][ y ]` является логической ошибкой. Компилятор C интерпретирует обращение `a[ x, y ]` как `a[ y ]` (потому что запятая в данном контексте интерпретируется как оператор запятой), то есть данная ошибка программиста не считается синтаксической ошибкой.

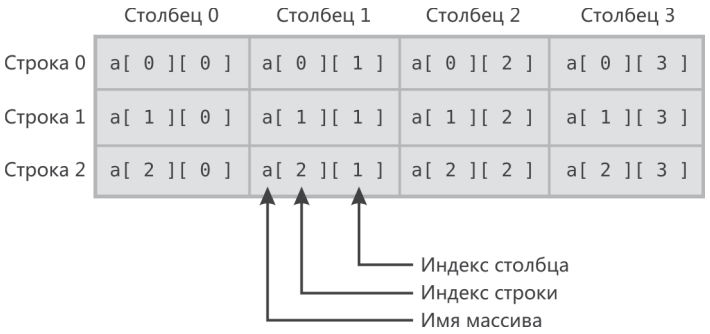


Рис. 6.2 | Двумерный массив с тремя строками и четырьмя столбцами

Многомерный массив можно инициализировать в инструкции объявления, подобно одномерному массиву. Например, двумерный массив `int b[2][2]` можно объявить и инициализировать инструкцией

```
int b[ 2 ][ 2 ] = { { 1, 2 }, { 3, 4 } };
```

Значения в фигурных скобках группируются построчно. Значения в первой паре фигурных скобок инициализируют элементы строки 0, значения во второй паре фигурных скобок – элементы строки 1. То есть в инструкции выше значения 1 и 2 инициализируют элементы `b[0][0]` и `b[0][1]` соответственно, а значения 3 и 4 – элементы `b[1][0]` и `b[1][1]`. *Если количество инициализирующих значений в строке меньше количества элементов, остальные элементы будут инициализированы нулями.* То есть объявление

```
int b[ 2 ][ 2 ] = { { 1 }, { 3, 4 } };
```

инициализирует элемент `b[0][0]` значением `1`, элемент `b[0][1]` – значением `0`, элемент `b[1][0]` – значением `3` и элемент `b[1][1]` – значением `4`. В примере 6.18 демонстрируются инструкции определения и инициализации двумерных массивов.

Программа определяет три массива по две строки и три столбца (шесть элементов) в каждом. Инструкция объявления массива `array1` (строка 11) включает все шесть инициализирующих значений. Значения 1, 2 и 3 в первой паре фигурных скобок инициализируют элементы строки 0; значения 4, 5 и 6 во второй паре фигурных скобок инициализируют элементы строки 1.

Пример 6.18 | Инициализация многомерных массивов

```
 1 // Пример 6.18: fig06_21.c
 2 // Инициализация многомерных массивов.
 3 #include <stdio.h>
 4
 5 void printArray( int a[][ 3 ] ); // прототип функции
 6
 7 // выполнение программы начинается с функции main
 8 int main( void )
 9 {
10    // инициализировать массивы array1, array2, array3
11    int array1[ 2 ][ 3 ] = { { 1, 2, 3 }, { 4, 5, 6 } };
12    int array2[ 2 ][ 3 ] = { 1, 2, 3, 4, 5 };
13    int array3[ 2 ][ 3 ] = { { 1, 2 }, { 4 } };
14
15    puts( "Values in array1 by row are:" );
16    printArray( array1 );
17
18    puts( "Values in array2 by row are:" );
19    printArray( array2 );
20
21    puts( "Values in array3 by row are:" );
22    printArray( array3 );
23 } // конец main
24
25 // выводит массив в две строки и три столбца
26 void printArray( int a[][ 3 ] )
27 {
28    size_t i; // счетчик строк
```

```
29    size_t j; // счетчик столбцов
30
31    // выполнить итерации по строкам
32    for ( i = 0; i <= 1; ++i ) {
33    // вывести значения в столбцах
34       for ( j = 0; j <= 2; ++j ) {
35          printf( "%d ", a[ i ][ j ] );
36       } // конец внутреннего цикла for
37       printf( "\n" ); // перейти на новую строку
38    } // конец внешнего цикла for
39 } // конец функции printArray
```

```
Values in array1 by row are:
1 2 3
4 5 6
Values in array2 by row are:
1 2 3
4 5 0
Values in array3 by row are:
1 2 0
4 0 0
```

Если из списка инициализирующих значений в объявлении `array1` убрать вложенные фигурные скобки, ограничивающие строки массива, компилятор выполнит инициализацию элементов по порядку, сначала в первой строке, затем во второй. Определение массива `array2` (строка 12) содержит пять инициализирующих значений. Инициализация будет выполняться в том же порядке – сначала будут инициализированы элементы первой строки, а потом второй. Элементы, для которых *отсутствует* явно указанное инициализирующее значение, будут инициализированы нулями, то есть элемент `array2[1][2]` будет инициализирован значением 0.

В определении массива `array3` (строка 13) указаны три инициализирующих значения, сгруппированных в две строки. Первая группа *явно* инициализирует первые два элемента первой строки значениями 1 и 2. Третий элемент будет инициализирован *нулем*. Вторая группа явно инициализирует первый элемент значением 4. Два последних элемента этой строки будут инициализированы нулями.

Для вывода элементов массивов программа вызывает функцию `printArray` (строки 26–41). Параметр-массив в определении функции объявляется как `const int a[][3]`. При объявлении параметра как одномерного массива квадратные скобки остаются *пустыми*. При передаче многомерного массива также не требуется указывать размерность в первой паре скобок, но все последующие измерения должны указываться обязательно. Компилятор будет использовать их для определения местоположения элементов многомерных массивов. Все элементы массивов хранятся в памяти последовательно, независимо от количества измерений. Двумерный массив хранится построчно – сначала элементы первой строки, потом второй и т. д.

Передача значений размерностей в объявлениях параметров дает компилятору возможность сообщить функции, как определяются адреса эле-

ментов массива. В двумерном массиве каждая строка, по сути, является одномерным массивом. Чтобы определить местоположение определенной строки, компилятор должен знать количество элементов в каждой строке, дабы перешагивать через блоки памяти при обращении к элементам массива. Так, при обращении к элементу `a[1][2]` в нашем примере компилятор знает: чтобы перейти ко второй строке, он должен перешагнуть через три элемента, составляющих первую строку, и затем обратиться к элементу 2 во второй строке.

Многие типичные операции с массивами выполняются с применением инструкций повторения. Например, следующая инструкция обнулит элементы строки 2 в массиве на рис. 6.2:

```
for ( column = 0; column <= 3; ++column ) {
   a[ 2 ][ column ] = 0;
}
```

Мы выбрали строку 2, поэтому первый индекс всегда будет равен 2. В цикле изменяется только второй индекс (определяющий столбец). Предыдущая инструкция `for` эквивалентна следующим инструкциям присваивания:

```
a[ 2 ][ 0 ] = 0;
a[ 2 ][ 1 ] = 0;
a[ 2 ][ 2 ] = 0;
a[ 2 ][ 3 ] = 0;
```

Следующие вложенные инструкции `for` вычисляют общую сумму значений элементов массива `a`.

```
total = 0;
for ( row = 0; row <= 2; ++row ) {
   for ( column = 0; column <= 3; ++column ) {
      total += a[ row ][ column ];
   }
}
```

Внутренний цикл вычисляет сумму элементов одной строки. Внешний цикл `for` сначала устанавливает номер строки `row` равным 0, чтобы внутренний цикл мог вычислить сумму элементов в этой строке. Затем он увеличивает номер строки на 1, чтобы внутренний цикл прибавил к сумме значения элементов второй строки. Потом он снова увеличивает номер строки на 1, и внутренний цикл прибавляет к сумме значения элементов следующей, третьей строки. После завершения внешней инструкции `for` переменная `total` будет содержать сумму значений всех элементов массива `a`.

Операции с двумерными массивами

В примере 6.19 демонстрируется реализация некоторых других типичных операций с массивами, выполняемых с использованием инструкций `for`. Каждая строка в массиве `studentGrades` размером 3 на 4 и каждый столбец представляют оценку, полученную тем или иным студентом на одном из четырех экзаменов в течение семестра. Операции с массивом реализованы

в виде четырех функций. Функция `minimum` (строки 41–60) определяет самую низкую оценку в массиве. Функция `maximum` (строки 63–82) определяет самую высокую оценку в массиве. Функция `average` (строки 85–96) определяет среднюю оценку для заданного студента за семестр. Функция `printArray` (строки 99–118) выводит двумерный массив в табличной форме.

Пример 6.19 | Операции с двумерным массивом

```
 1 // Пример 6.19: fig06_22.c
 2 // Операции с двумерным массивом.
 3 #include <stdio.h>
 4 #define STUDENTS 3
 5 #define EXAMS 4
 6
 7 // прототипы функций
 8 int minimum( int grades[][ EXAMS ], size_t pupils, size_t tests );
 9 int maximum( int grades[][ EXAMS ], size_t pupils, size_t tests );
10 double average( const int setOfGrades[], size_t tests );
11 void printArray( int grades[][ EXAMS ], size_t pupils, size_t tests );
12
13 // выполнение программы начинается с функции main
14 int main( void )
15 {
16    size_t student; // счетчик студентов
17
18    // инициализировать массив оценок для трех студентов
19    int studentGrades[ STUDENTS ][ EXAMS ] =
20       { { 77, 68, 86, 73 },
21         { 96, 87, 89, 78 },
22         { 70, 90, 86, 81 } };
23
24    // вывести массив studentGrades
25    puts( "The array is:" );
26    printArray( studentGrades, STUDENTS, EXAMS );
27
28    // определить низшую и высшую оценки
29    printf( "\n\nLowest grade: %d\nHighest grade: %d\n",
30       minimum( studentGrades, STUDENTS, EXAMS ),
31       maximum( studentGrades, STUDENTS, EXAMS ) );
32
33    // вычислить среднюю оценку для каждого студента в отдельности
34    for ( student = 0; student < STUDENTS; ++student ) {
35       printf( "The average grade for student %u is %.2f\n",
36          student, average( studentGrades[ student ], EXAMS ) );
37    } // конец for
38 } // конец main
39
40 // Находит низшую оценку
41 int minimum( int grades[][ EXAMS ], size_t pupils, size_t tests )
42 {
43    size_t i; // счетчик студентов
44    size_t j; // счетчик экзаменов
45    int lowGrade = 100; // инициализировать наивысшей возможной оценкой
46
47    // обойти строки в массиве оценок
48    for ( i = 0; i < pupils; ++i ) {
```

```
49
50       // обойти столбцы в массиве оценок
51       for ( j = 0; j < tests; ++j ) {
52
53          if ( grades[ i ][ j ] < lowGrade ) {
54             lowGrade = grades[ i ][ j ];
55          } // конец if
56       } // конец внутреннего цикла for
57    } // конец внешнего цикла for
58
59    return lowGrade; // вернуть низшую оценку
60 } // конец функции minimum
61
62 // Находит высшую оценку
63 int maximum( int grades[][ EXAMS ], size_t pupils, size_t tests )
64 {
65    size_t i; // счетчик студентов
66    size_t j; // счетчик экзаменов
67    int highGrade = 0; // инициализировать низшей возможной оценкой
68
69    // обойти строки в массиве оценок
70    for ( i = 0; i < pupils; ++i ) {
71
72       // обойти столбцы в массиве оценок
73       for ( j = 0; j < tests; ++j ) {
74
75          if ( grades[ i ][ j ] > highGrade ) {
76             highGrade = grades[ i ][ j ];
77          } // конец if
78       } // конец внутреннего цикла for
79    } // конец внешнего цикла for
80
81    return highGrade; // вернуть высшую оценку
82 } // конец функции maximum
83
84 // Определяет среднюю оценку за семестр для заданного студента
85 double average( const int setOfGrades[], size_t tests )
86 {
87    size_t i;      // счетчик экзаментов
88    int total = 0; // сумма оценок
89
90    // вычислить сумму всех оценок данного студента
91    for ( i = 0; i < tests; ++i ) {
92       total += setOfGrades[ i ];
93    } // конец for
94
95    return ( double ) total / tests; // среднее
96 } // конец функции average
97
98 // Выводит массив
99 void printArray( int grades[][ EXAMS ], size_t pupils, size_t tests )
100 {
101    size_t i; // счетчик студентов
102    size_t j; // счетчик экзаменов
103
104    // вывести заголовки столбцов
105    printf( "%s", "                [0] [1] [2] [3]" );
```

```
106
107    // вывести оценки в табличной форме
108    for ( i = 0; i < pupils; ++i ) {
109
110       // вывести метки строк
111       printf( "\nstudentGrades[%d] ", i );
112
113       // вывести оценки для одного студента
114       for ( j = 0; j < tests; ++j ) {
115          printf( "%-5d", grades[ i ][ j ] );
116       } // конец внутреннего цикла for
117    } // конец внешнего цикла for
118 } // конец функции printArray
```

```
The array is:
                 [0] [1] [2] [3]
studentGrades[0] 77  68  86  73
studentGrades[1] 96  87  89  78
studentGrades[2] 70  90  86  81

Lowest grade: 68
Highest grade: 96
The average grade for student 0 is 76.00
The average grade for student 1 is 87.50
The average grade for student 2 is 81.75
```

Функции `minimum`, `maximum` и `printArray` принимают три аргумента – массив `studentGrades` (внутри функций называется `grades`), количество студентов (строк в массиве) и количество экзаменов (количество столбцов). Каждая функция выполняет обход оценок в массиве с помощью вложенных инструкций `for`. Ниже приводится вложенная инструкция `for` из определения функции `minimum`:

```
47    // обойти строки в массиве оценок
48    for ( i = 0; i < pupils; ++i ) {
49
50       // обойти столбцы в массиве оценок
51       for ( j = 0; j < tests; ++j ) {
52
53          if ( grades[ i ][ j ] < lowGrade ) {
54             lowGrade = grades[ i ][ j ];
55          } // конец if
56       } // конец внутреннего цикла for
57    } // конец внешнего цикла for
```

Внешний цикл `for` начинается с установки переменной `i` (то есть индекса строки) в значение 0, чтобы обеспечить сравнение элементов этой строки (оценок первого студента) с переменной `lowGrade` в теле внутреннего цикла `for`. Внутренний цикл `for` выполняет обход элементов (оценок) данной строки и сравнивает каждую из них с переменной `lowGrade`. Если текущая оценка оказывается меньше `lowGrade`, она запоминается в переменной `lowGrade`. Затем инструкция `for` внешнего цикла увеличивает индекс строки до 1. Внутренний цикл `for` сравнивает оценки в этой строке с переменной

lowGrade. Внешняя инструкция for вновь увеличивает индекс строки до 2. И снова внутренний цикл for сравнивает оценки в ней с переменной lowGrade. После завершения *вложенных* друг в друга инструкций for в переменной lowGrade оказывается самая низкая оценка из числа имеющихся в двумерном массиве. Функция maximum действует подобно функции minimum.

Функция average (строки 85–96) принимает два аргумента – одномерный массив setOfGrades с экзаменационными оценками одного студента и количество оценок в массиве. В первом аргументе функции average передается studentGrades[student] – адрес строки в двумерном массиве. То есть аргумент studentGrades[1] соответствует адресу первого элемента в строке с индексом 1. Не забывайте, что двумерный массив, по сути, является одномерным массивом массивов, а имя одномерного массива соответствует адресу первого элемента этого массива в памяти. Функция average вычисляет сумму значений элементов массива, делит ее на количество оценок и возвращает вещественное число, представляющее результат.

6.10 Массивы переменной длины

В ранних версиях языка C поддерживались только массивы фиксированной длины. Но как быть, если к моменту компиляции размер массива не известен? Чтобы решить эту проблему, необходимо было прибегать к помощи функций для работы с динамической памятью, таких как malloc и др. Позднее в стандарт языка C была включена поддержка массивов переменной длины (Variable-Length Arrays, VLA). Однако, несмотря на название, эти массивы не могут изменять своих размеров, иначе это поставило бы под угрозу целостность данных, находящихся в памяти по соседству с ними. **Массив переменной длины** – это массив, длина (или размер) которого определяется во время выполнения, но только один раз. Программа в примере 6.20 объявляет несколько массивов переменной длины и выводит их содержимое. [*Обратите внимание*: массивы переменной длины не поддерживаются в Microsoft Visual C++.]

Пример 6.20 | Массивы переменной длины

```
 1 // Пример 6.20: fig06_14.c
 2 // Массивы переменной длины
 3 #include <stdio.h>
 4
 5 // прототипы функций
 6 void print1DArray( int size, int arr[ size ] );
 7 void print2DArray( int row, int col, int arr[ row ][ col ] );
 8
 9 int main( void )
10 {
11    int arraySize;              // размер 1-мерного массива
12    int row1, col1, row2, col2; // размеры 2-мерных массивов
13
14    printf( "%s", "Enter size of a one-dimensional array: " );
```

```
15    scanf( "%d", &arraySize );
16
17    printf( "%s", "Enter number of rows and columns in a 2-D array: " );
18    scanf( "%d %d", &row1, &col1 );
19
20    printf( "%s",
21       "Enter number of rows and columns in another 2-D array: " );
22    scanf( "%d %d", &row2, &col2 );
23
24    int array[ arraySize ];       // 1-мерный массив переменной длины
25    int array2D1[ row1 ][ col1 ]; // 2-мерный массив переменной длины
26    int array2D2[ row2 ][ col2 ]; // 2-мерный массив переменной длины
27
28    // проверить действие оператора sizeof с массивом переменной длины
29    printf( "\nsizeof(array) yields array size of %d bytes\n",
30       sizeof( array ) );
31
32    // присвоить значения элементам 1-мерного массива переменной длины
33    for ( int i = 0; i < arraySize; ++i ) {
34       array[ i ] = i * i;
35    } // конец for
36
37    // присвоить значения элементам первого 2-мерного массива
38    for ( int i = 0; i < row1; ++i ) {
39       for ( int j = 0; j < col1; ++j ) {
40          array2D1[ i ][ j ] = i + j;
41       } // конец for
42    } // конец for
43
44    // присвоить значения элементам второго 2-мерного массива
45    for ( int i = 0; i < row2; ++i ) {
46       for ( int j = 0; j < col2; ++j ) {
47          array2D2[ i ][ j ] = i + j;
48       } // конец for
49    } // конец for
50
51    puts( "\nOne-dimensional array:" );
52    print1DArray( arraySize, array ); // вывести 1-мерный массив
53
54    puts( "\nFirst two-dimensional array:" );
55    print2DArray( row1, col1, array2D1 ); // вывести 2-мерный массив
56
57    puts( "\nSecond two-dimensional array:" );
58    print2DArray( row2, col2, array2D2 ); // вывести второй 2-мерный массив
59 } // конец main
60
61 void print1DArray( int size, int array[ size ] )
62 {
63    // вывести содержимое массива
64    for ( int i = 0; i < size; i++ ) {
65       printf( "array[%d] = %d\n", i, array[ i ] );
66    } // конец for
67 } // конец функции print1DArray
68
69 void print2DArray( int row, int col, int arr[ row ][ col ] )
70 {
71    // вывести содержимое массива
```

```
72    for ( int i = 0; i < row; ++i ) {
73       for ( int j = 0; j < col; ++j ) {
74          printf( "%5d", arr[ i ][ j ] );
75       } // конец for
76
77       puts( "" );
78    } // конец for
79 } // конец функции print2DArray
```

```
Enter size of a one-dimensional array: 6
Enter number of rows and columns in a 2-D array: 2 5
Enter number of rows and columns in another 2-D array: 4 3

sizeof(array) yields array size of 24 bytes

One-dimensional array:
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25

First two-dimensional array:
    0    1    2    3    4
    1    2    3    4    5

Second two-dimensional array:
    0    1    2
    1    2    3
    2    3    4
    3    4    5
```

Первым делом мы предлагаем пользователю ввести размеры одномерного и двух двумерных массивов (строки 14–22). Затем в строках 24–26 производится объявление массивов переменной длины с соответствующими размерами. Такое объявление допустимо, если переменные, представляющие размеры, имеют целочисленный тип.

После объявления массивов мы пробуем применить оператор `sizeof` в строках 29–30, чтобы убедиться, что массив переменной длины имеет нужный размер. В прежних версиях языка C оператор `sizeof` всегда выполнялся на этапе компиляции, но при применении к массивам переменной длины он определяет результат на этапе выполнения. В окне с выводом программы можно видеть, что оператор `sizeof` вернул размер, равный 24 байтам – количество элементов по четыре байта в каждом, потому что значение типа `int` на нашей машине занимает именно 4 байта.

Далее мы присваиваем значения элементам массивов переменной длины (строки 33–49). В качестве условия продолжения цикла при заполнении одномерного массива мы используем выражение `i < arraySize`. Как и при работе с массивами фиксированной длины, *никакой защиты от выхода за пределы массива не предусматривается.*

В строках 61–67 определяется функция `print1DArray`, принимающая одномерный массив переменной длины. Синтаксис определения параметра, являющегося массивом переменной длины, ничем не отличается от определения параметра, являющегося массивом фиксированной длины. В объявлении параметра `array` мы использовали переменную `size`, но никаких проверок компилятором здесь не производится, кроме того что он убеждается, что эта переменная имеет целочисленный тип, – это сделано исключительно в целях документирования кода для других программистов.

Функция `print2DArray` (строки 69–79) принимает двумерный массив переменной длины и выводит его содержимое на экран. В разделе 6.9 уже говорилось, что в объявлениях параметров функций, являющихся многомерными массивами, должны явно указываться все размерности, кроме первой. То же требование действует и для многомерных массивов переменной длины, за исключением того, что размеры могут определяться переменными. Начальное значение параметра `col`, что передается функции, используется для преобразования двумерных индексов в смещения внутри непрерывной области памяти, где хранится массив, как и в случае с массивами переменной длины. Изменение значения переменной `col` внутри функции никак не скажется на индексации элементов, но если передать ошибочное значение, последствия могут быть самыми плачевными.

6.11 Безопасное программирование на C

Проверка границ массивов

При обращении к элементам массива очень важно гарантировать, что используемые индексы не выходят за границы массива – то есть индексы должны быть больше или равны 0 и меньше количества элементов в массиве. Индексы строк и столбцов в двумерных массивах должны быть больше или равны 0 и меньше количества строк и столбцов соответственно. Это правило распространяется и на массивы с дополнительными размерностями.

Неконтролируемая возможность читать или писать данные в элементы за границами массива является серьезной угрозой безопасности. Чтение значений элементов за пределами массива может вызвать аварийное завершение программы или неправильную ее работу. Запись в элементы за границами массива (эта проблема известна как *переполнение буфера*) может вызвать повреждение данных в памяти программы, аварийное ее завершение или даже открыть злоумышленникам доступ к системе и возможность запускать вредоносный код.

Как отмечалось в разделе 6.4, в языке C *отсутствуют механизмы автоматической проверки на выход за границы массивов*, поэтому вы сами должны предоставлять такие механизмы. Приемы, позволяющие предотвратить упомянутые проблемы, описаны в руководстве ARR30-C центра CERT по адресу: `www.securecoding.cert.org`.

scanf_s

Проверка границ играет не менее важную роль и при работе со строками. Выполняя чтение строки в массив символов, функция `scanf` *не* предотвращает переполнение буфера. Если количество введенных символов окажется больше или равно размеру буфера, `scanf` выполнит запись символов – включая завершающий нулевой символ (`'\0'`) – за границы массива. Это может привести к затиранию данных в других переменных и, при определенных условиях, к затиранию нулевого символа `'\0'` в строковых переменных, находящихся в памяти по соседству.

Функции определяют конец строки, отыскивая завершающий символ `'\0'`. Например, функция `printf` выводит строку, последовательно читая символы из памяти, пока не встретит символ `'\0'`. Если этот символ отсутствует, `printf` будет продолжать читать символы, пока не встретит завершающий символ `'\0'` где-нибудь в памяти.

Приложение Annex K к стандарту языка C ввело новые, более безопасные версии многих функций для работы со строками и ввода/вывода, включая `scanf_s` – версию функции `scanf`, которая выполняет дополнительные проверки, чтобы *не выйти* за границы массива символов, используемого для сохранения введенной строки. Если предположить, что массив `myString` имеет емкость в 20 символов, следующая инструкция:

```
scanf_s( "%19s", myString, 20 );
```

обеспечит безопасное чтение строки в массив `myString`. Функция `scanf_s` требует указывать по *два* аргумента *для каждого* спецификатора `%s` в строке формата – массив символов, куда должна быть помещена введенная строка, и количество элементов в этом массиве. Второй из этих аргументов используется функцией `scanf_s` для предотвращения переполнения буфера. Например, в спецификаторе `%s` можно указать слишком большую ширину поля, превышающую размер символьного массива, отведенного для строки, или вообще опустить ее. Если количество введенных символов (плюс завершающий нулевой символ) окажется больше количества элементов в массиве, преобразование `%s` потерпит неудачу. Так как инструкция выше содержит единственный спецификатор в строке формата, `scanf_s` может вернуть `0`, сообщая, что преобразование не было выполнено и содержимое `myString` не изменилось.

Вообще говоря, если ваш компилятор поддерживает функции, определяемые приложением Annex K к стандарту языка C, вы обязательно должны использовать их. Другие функции, определяемые приложением Annex K, мы будем обсуждать в последующих разделах «Безопасное программирование на C».

Не используйте строки, вводимые пользователями, в качестве строк управления форматом

Возможно, вы обратили внимание, что нигде в этой книге мы не используем версию функции `printf` с одним аргументом. Мы всегда применяем одну из следующих форм:

- когда требуется вывести `'\n'` после строки, мы используем функцию `puts` (которая автоматически выводит `'\n'` после своего единственного строкового аргумента, например:

```
puts( "Welcome to C!" );
```

- когда требуется оставить текстовый курсор в той же строке, мы используем функцию `printf`, например:

```
printf( "%s", "Enter first integer: " );
```

Для вывода простых строковых литералов мы, конечно, могли бы использовать форму вызова `printf` с одним аргументом:

```
printf( "Welcome to C!\n" );
printf( "Enter first integer: " );
```

Когда функция `printf` интерпретирует строку управления форматом в своем первом (и возможно, единственном) аргументе, она выполняет некоторые операции, в зависимости от спецификаторов преобразований в этой строке. Если в качестве строки формата используется строка, полученная от пользователя, злоумышленник может подсунуть специально подготовленные спецификаторы преобразований, которые будут «выполнены» функцией форматированного вывода. Теперь, когда вы знаете, как читать строки в массивы символов, важно заметить, что вы также никогда не должны использовать в качестве строк управления форматом массивы символов, которые могут содержать ввод пользователя. За дополнительной информацией обращайтесь к руководству FIO30-C центра CERT по адресу: `www.securecoding.cert.org`.

7

Указатели

В этой главе вы:

- познакомитесь с указателями и операторами указателей;
- научитесь пользоваться указателями для передачи аргументов функциям по ссылке;
- узнаете о тесной связи между указателями, массивами и строками;
- познакомитесь с приемами использования указателей на функции;
- научитесь определять и использовать массивы строк.

7.1 Введение

В этой главе мы обсудим одну из самых мощных особенностей языка программирования C – **указатели**[1]. Указатели также относятся к одной из самых сложных тем. Применение указателей позволяет реализовать передачу аргументов функциям по ссылке, передачу функций между функциями, а также создавать динамические структуры данных и управлять ими (которые могут увеличиваться и уменьшаться в размерах во время выполнения), такие как связанные списки, очереди, стеки и деревья. Данная глава описывает основные понятия, связанные с указателями. В главе 10 мы займемся исследованием применения указателей для работы со структурами. В главе 12 будут описаны приемы динамического управления памятью и представлены примеры создания и использования динамических структур данных.

7.2 Переменные-указатели, определение и инициализация

Указатели – это переменные, значениями которых являются *адреса в памяти*. Обычные переменные содержат непосредственные значения. Указатель, напротив, содержит адрес переменной, хранящей непосредственное значе-

[1] Указатели и другие конструкции, основанные на указателях, такие как массивы и строки, при неправильном использовании (случайно или преднамеренно) могут привести к ошибкам в работе программ и появлению брешей в системе безопасности. Этой важной теме посвящено множество книг, статей и сообщений, которые можно найти в нашем центре ресурсов «Secure C Programming Resource Center» (`www.deitel.com/SecureC/`).

ние. В этом смысле имя переменной является *непосредственной* ссылкой на значение, а указатель – *косвенной* (рис. 7.1). Доступ к значению через указатель называется **косвенным доступом**.

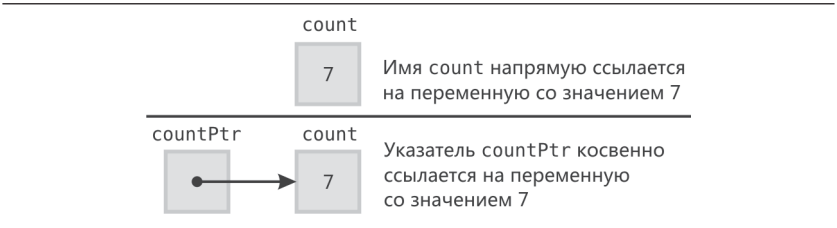


Рис. 7.1 | Прямые и косвенные ссылки на значения

Объявление указателей

Указатели, как и обычные переменные, должны определяться, прежде чем их можно будет использовать. Следующее определение:

```
int *countPtr, count;
```

объявляет переменную типа `int*` (то есть указатель на целое число) и читается как «`countPtr` – указатель на значение типа `int`» или «`countPtr` указывает на объект типа `int`». Здесь также объявляется переменная `count` типа `int`, а не указатель на значение типа `int`. Спецификатор `*` применяется только к имени `countPtr` в определении. Когда символ `*` используется в определении таким способом, он сообщает, что определяемая переменная является указателем. Указатели могут указывать на объекты любых типов. Чтобы исключить неоднозначность при определении переменных-указателей и обычных переменных в одной инструкции, в одном объявлении всегда следует определять только одну переменную.

*Звездочка (*), используемая для объявления переменной-указателя, не распространяет свое действие на все имена в объявлении. Каждый указатель должен объявляться с собственным префиксом * в имени, например если требуется объявить переменные `xPtr` и `yPtr` указателями на значения типа `int`, они должны объявляться как `int *xPtr, *yPtr;`.*

Мы предпочитаем добавлять в имена переменных-указателей последовательность символов `Ptr`, чтобы было видно, что они являются указателями и должны обрабатываться соответственно.

Инициализация указателей и присваивание им значений

Указатели могут инициализироваться в момент объявления, а также им можно присваивать значения уже в ходе выполнения. Указатель может быть

инициализирован значением NULL, 0 или адресом. Указатель со значением NULL указывает *в никуда*. NULL – это *символическая константа*, объявленная в заголовочном файле <stddef.h> (ее объявление также можно найти в нескольких других заголовочных файлах, таких как <stdio.h>). Инициализация указателя значением 0 эквивалентна инициализации значением NULL, но вообще предпочтительнее использовать NULL. Когда указателю присваивается значение 0, оно сначала преобразуется в указатель соответствующего типа. Значение 0 – *единственное* целочисленное значение, которое может быть присвоено переменной-указателю непосредственно. О присваивании адресов переменных указателям рассказывается в разделе 7.3.

Всегда инициализируйте указатели, чтобы избежать неожиданностей.

7.3 Операторы указателей

Амперсанд (&), или оператор взятия адреса, – это унарный оператор, возвращающий адрес своего операнда. Например, допустим, что имеются следующие определения:

```
int y = 5;
int *yPtr;
```

Тогда инструкция

```
yPtr = &y;
```

присвоит переменной-указателю yPtr *адрес* переменной y. После этого о переменной yPtr можно сказать, что она «указывает на» переменную y. На рис. 7.2 показано схематическое представление памяти после выполнения присваивания.

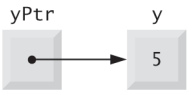


Рис. 7.2 | Графическое представление указателя, указывающего на целочисленную переменную

Представление указателя в памяти

На рис. 7.3 показано представление указателя в памяти, где предполагается, что целочисленная переменная y хранится в памяти по адресу 600000, а переменная-указатель yPtr – по адресу 500000. Операнд оператора взятия адреса обязательно должен быть переменной; этот оператор не может применяться к константам и выражениям.

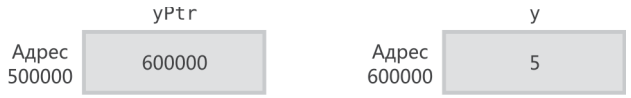


Рис. 7.3 | Представление переменных y и yPtr в памяти

Оператор косвенного обращения (*)

Унарный оператор *, часто называемый **оператором косвенного обращения** или **оператором разыменования**, возвращает значение объекта, на который указывает его операнд (то есть указатель). Например, инструкция

```
printf( "%d", *yPtr );
```

выведет значение переменной y, а именно число 5. Такой способ использования унарного оператора * называется **разыменованием указателя**.

Разыменование неинициализированного указателя или указателя, которому не был присвоен действительный адрес переменной в памяти, является ошибкой. Это может привести к аварийному завершению приложения или к повреждению важных данных и получению ошибочных результатов.

*Демонстрация операторов & и **

В примере 7.1 демонстрируется использование операторов указателей & и *. Спецификатор преобразования `%p` применяется в вызове функции `printf` для вывода адресов в памяти в *шестнадцатеричном* виде. (Дополнительная информация о шестнадцатеричных числах приводится в приложении C.) Обратите внимание, что *адрес* переменной `a` совпадает со *значением* переменной `aPtr`, это подтверждает, что переменной `aPtr` действительно присвоен адрес (строка 11). Операторы & и * дополняют друг друга – когда они оба последовательно применяются к переменной `aPtr` в любом порядке (строка 21), выводится один и тот же результат. В табл. 7.1 указаны приоритеты и ассоциативность операторов, с которыми мы познакомились к данному моменту.

Пример 7.1 | Использование операторов указателей & и *

```
 1 // Пример 7.1: fig07_04.c
 2 // Использование операторов указателей & и *.
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    int a; // a - целочисленная переменная
 8    int *aPtr; // aPtr - указатель на целочисленную переменную
 9
10    a = 7;
11    aPtr = &a; // присвоить переменной aPtr адрес переменной a
```

```
12
13    printf( "The address of a is %p"
14            "\nThe value of aPtr is %p", &a, aPtr);
15
16    printf( "\n\nThe value of a is %d"
17            "\nThe value of *aPtr is %d", a, *aPtr);
18
19    printf( "\n\nShowing that * and & are complements of "
20            "each other\n&*aPtr = %p"
21            "\n*&aPtr = %p\n", &*aPtr, *&aPtr);
22 } // конец main
```

```
The address of a is 0028FEC0
The value of aPtr is 0028FEC0

The value of a is 7
The value of *aPtr is 7

Showing that * and & are complements of each other
&*aPtr = 0028FEC0
*&aPtr = 0028FEC0
```

Таблица 7.1 | Приоритеты и ассоциативность операторов

Операторы	Ассоциативность	Тип
`[] ()` ++ (*постфиксный*) -- (*постфиксный*)	Справа налево	Постфиксный
`+ - ++ -- ! * &` (*тип*)	Справа налево	Унарный
`* / %`	Слева направо	Мультипликативный
`+ -`	Слева направо	Аддитивный
`< <= > >=`	Слева направо	Проверка отношения
`== !=`	Слева направо	Проверка на равенство
`&&`	Слева направо	Логическое И
`||`	Слева направо	Логическое ИЛИ
`?:`	Справа налево	Условный
`= += -= *= /= %=`	Справа налево	Присваивание
`,`	Слева направо	Запятая

7.4 Передача аргументов функциям по ссылке

Существуют два способа передачи аргументов функциям – **по значению** и **по ссылке**. *Все аргументы в языке C передаются по значению.* Как было показано в главе 5, для возврата значения (или возврата управления без возврата значения) вызывающей функции из вызываемой можно использовать инструкцию `return`. Часто бывает необходимо иметь возможность изменять переменные, объявленные в вызывающей функции, или передавать указатель на большой объект данных, чтобы избежать накладных расходов на ко-

пирование этого объекта при передаче по значению (что требует времени и памяти для хранения копии объекта).

В языке C указатели и оператор косвенного доступа используются для имитации передачи аргументов по ссылке. Когда функция должна изменить значение аргумента, ей передается адрес этого аргумента. Обычно это реализуется применением оператора взятия адреса (&) к переменной (объявленной в вызывающей функции), значение которой должно быть изменено. Как было показано в главе 6, массивы передаются функциям без применения оператора &, потому что компилятор языка C автоматически передает массив как адрес первого его элемента (имя массива действует эквивалентно выражению `&arrayName[0]`). Когда функции передается адрес переменной, внутри нее можно использовать оператор косвенного доступа (*), чтобы изменить значение, хранящееся по этому адресу.

Передача по значению

Программы в примерах 7.2 и 7.3 демонстрируют две версии функции, возводящей указанное целочисленное значение в третью степень, – `cubeByValue` и `cubeByReference`. В примере 7.2 переменная передается функции `cubeByValue` (строка 14) по значению. Эта функция возводит значение аргумента в третью степень и возвращает полученный результат с помощью инструкции `return`. Новое значение затем присваивается переменной `number`, объявленной в функции `main` (строка 14).

Пример 7.2 | Возведение в третью степень с передачей аргумента по значению

```
 1 // Пример 7.2: fig07_06.c
 2 // Возведение в третью степень с передачей аргумента по значению.
 3 #include <stdio.h>
 4
 5 int cubeByValue( int n ); // прототип
 6
 7 int main( void )
 8 {
 9    int number = 5; // инициализировать переменную number
10
11    printf( "The original value of number is %d", number );
12
13    // передать number функции cubeByValue по значению
14    number = cubeByValue( number );
15
16    printf( "\nThe new value of number is %d\n", number );
17 } // конец main
18
19 // возводит целочисленный аргумент в третью степень и возвращает результат
20 int cubeByValue( int n )
21 {
22    return n * n * n; // возвести число в третью степень и вернуть результат
23 } // конец функции cubeByValue
```

```
The original value of number is 5
The new value of number is 125
```

Передача по ссылке

В примере 7.3 переменная `number` передается по ссылке (строка 15) – функции `cubeByReference` передается адрес переменной `number`. Функция `cubeByReference` принимает в единственном параметре указатель на целое число с именем `nPtr` (строка 21). Она *разыменовывает* указатель, возводит в третью степень число, на которое указывает переменная `nPtr` (строка 23), и присваивает результат `*nPtr` (фактически это переменная `number` в функции `main`), изменяя тем самым значение переменной `number`.

Пример 7.3 | Возведение в третью степень с передачей аргумента по ссылке

```
 1 // Пример 7.3: fig07_07.c
 2 // Возведение в третью степень с передачей аргумента по ссылке.
 3
 4 #include <stdio.h>
 5
 6 void cubeByReference( int *nPtr ); // прототип функции
 7
 8 int main( void )
 9 {
10    int number = 5; // инициализировать переменную number
11
12    printf( "The original value of number is %d", number );
13
14    // передать функции cubeByReference адрес переменной number
15    cubeByReference( &number );
16
17    printf( "\nThe new value of number is %d\n", number );
18 } // конец main
19
20 // возводит в третью степень значение *nPtr; фактически изменяет number
21 void cubeByReference( int *nPtr )
22 {
23    *nPtr = *nPtr * *nPtr * *nPtr; // возвести *nPtr в третью степень
24 }
```

```
The original value of number is 5
The new value of number is 125
```

Функция, принимающая адрес переменной в виде аргумента, должна определить параметр-указатель. Например, в примере 7.3 заголовок функции `cubeByReference` (строка 21) имеет следующий вид:

```
21 void cubeByReference( int *nPtr )
```

Данный заголовок указывает, что `cubeByReference` принимает адрес целочисленной переменной, сохраняет этот адрес в своей локальной переменной `nPtr` и ничего не возвращает.

Прототип функции `cubeByReference` (строка 6) содержит объявление `int *` в круглых скобках. Как и в случаях с параметрами других типов, *имя* параметра-указателя в прототипе функции можно *опустить*. Мы включи-

ли имена исключительно для документирования кода – они игнорируются компилятором.

Функции, принимающие одномерные массивы в аргументах

Если функция принимает в аргументе одномерный массив, в списке параметров в прототипе и в заголовке функции можно использовать форму записи указателей, как в заголовке функции `cubeByReference` (строка 21). Компилятор С не различает ситуации, когда функции принимают указатели и одномерные массивы. Однако сама функция должна «знать», что она принимает – одномерный массив или адрес единственной переменной. Когда компилятор встречает определение параметра в форме одномерного массива `int b[]`, он преобразует его в определение параметра указателя `int *b`. Эти две формы являются взаимозаменяемыми.

Используйте передачу аргументов по значению, если вызывающей функции явно не требуется, чтобы вызываемая функция изменяла значение передаваемой переменной. Этот прием предотвратит случайное изменение переменной и является еще одним примером использования принципа наименьших привилегий.

Графическое представление передачи аргумента по значению и по ссылке

На рис. 7.4 и 7.5 изображены диаграммы выполнения примеров 7.2 и 7.3 соответственно. На этих рисунках значения в прямоугольниках над выражениями или переменными отражают значения этих выражений или переменных. На обоих рисунках в диаграммах выполнения функций `cubeByValue` (рис. 7.4) и `cubeByReference` (рис. 7.5) значения справа действительны только в момент выполнения этих функций.

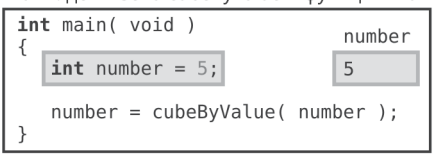
7.5 Использование квалификатора const с указателями

Квалификатор `const` дает возможность сообщить компилятору, что значение отмеченной им переменной не должно изменяться.

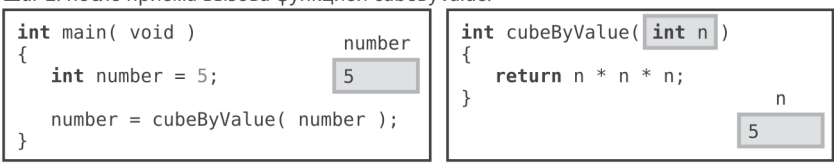
Квалификатор `const` можно использовать для принудительного проведения в жизнь принципа наименьших привилегий. Он поможет сократить время на отладку, устранить нежелательные побочные эффекты и упростить сопровождение и дальнейшее развитие программы.

За долгие годы накопился огромный объем кода, написанного на ранних версиях языка С, где отсутствовала поддержка квалификатора `const`. Вследствие этого появилась масса возможностей по усовершенствованию и реорганизации старого кода на С.

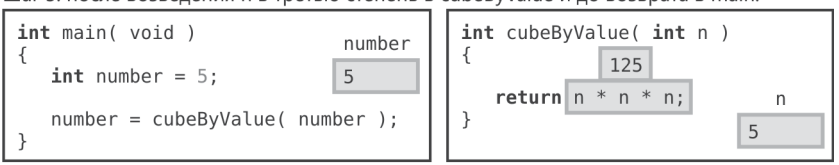
Шаг 1: до вызова cubeByValue в функции main:



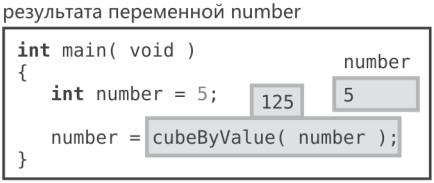
Шаг 2: после приема вызова функцией cubeByValue:



Шаг 3: после возведения n в третью степень в cubeByValue и до возврата в main:



Шаг 4: после возврата в main из cubeByValueи до присваивания результата переменной number



Шаг 5: после присваивания результата переменной number в main

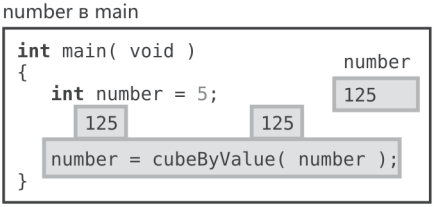


Рис. 7.4 | Передача аргумента по значению

Применение const к параметрам функций

Существует шесть вариантов применения (или неприменения) квалификатора к параметрам функций – два варианта к параметрам, передаваемым по значению, и четыре варианта к параметрам, передаваемым по ссылке. Как

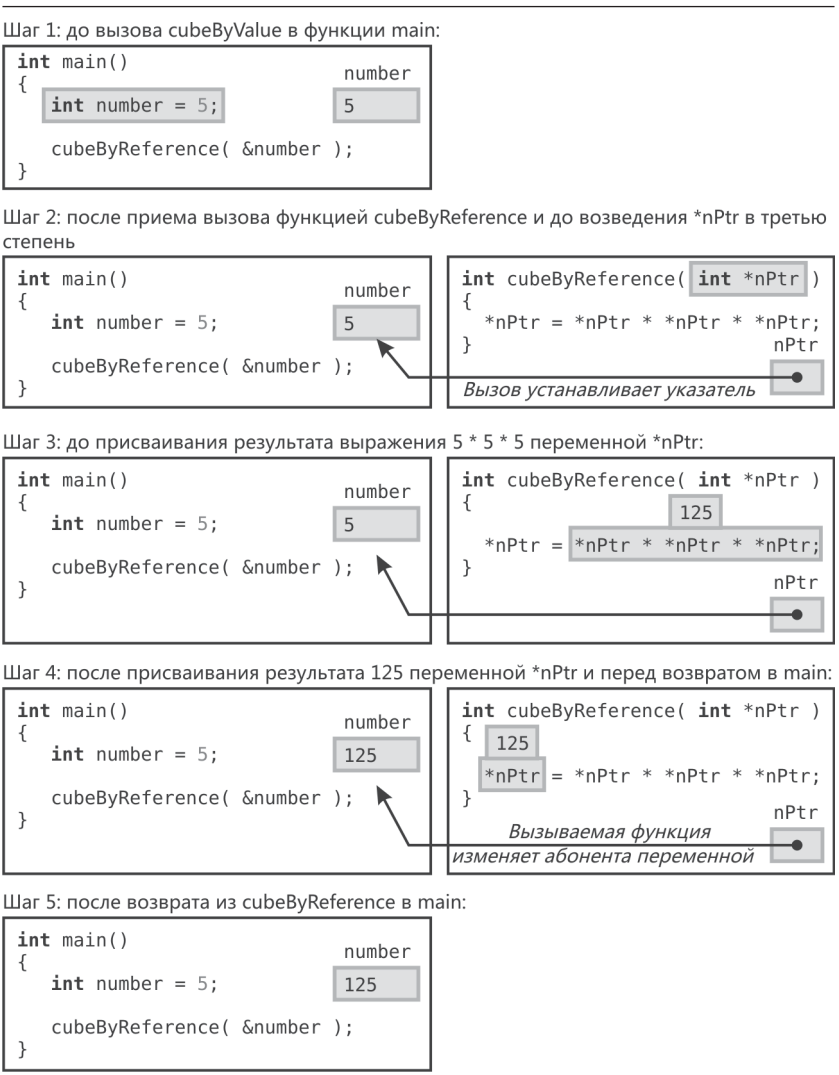


Рис. 7.5 | Передача аргумента по ссылке

правильно выбрать тот или иной вариант в каждом конкретном случае? Руководствуйтесь принципом наименьших привилегий.

Всегда предоставляйте функциям доступ к данным посредством параметров, достаточный для решения поставленной задачи, но не более того.

В главе 5 мы узнали, что в языке C *все аргументы передаются функциям по значению* – функции получают копии параметров. Если функция изменит копию, это никак *не* отразится на оригинальной переменной в вызывающей функции. Во многих случаях функции изменяют свои аргументы в процессе решения задачи. Однако в некоторых ситуациях значение *не* должно изменяться в вызываемой функции, даже при том, что аргумент является всего лишь *копией* оригинального значения.

Представьте функцию, принимающую в виде аргументов одномерный массив и его размер, которая выводит содержимое массива на экран. Такая функция должна в цикле обойти элементы массива и вывести их по отдельности. Аргумент с размером массива используется в теле функции для определения максимально допустимого индекса, чтобы цикл мог завершиться после вывода последнего элемента. Ни размер, ни содержимое массива не должны изменяться в теле функции.

Если значение некоторого аргумента не изменяется (или не должно изменяться) в теле функции, соответствующий параметр следует объявить с квалификатором `const`*, чтобы предупредить случайное его изменение.*

Если в функции будет предпринята попытка изменить значение аргумента, объявленного с квалификатором `const`, компилятор обнаружит ее и выведет либо предупреждение, либо сообщение об ошибке, в зависимости от используемой версии компилятора.

Попытка передать аргумент по значению функции, принимающей указатель, является ошибкой. Некоторые компиляторы передают значения, предполагая, что они являются указателями и выполняют разыменование. Во время выполнения это часто приводит к ошибкам нарушения прав доступа к памяти. Другие компиляторы обнаруживают несоответствие типов аргументов и параметров и генерируют сообщения об ошибках во время компиляции.

Существуют четыре способа передачи указателей в функции: передача **изменяемого указателя на изменяемые данные**, передача **константного указателя на изменяемые данные**, передача **изменяемого указателя на константные данные** и передача **константного указателя на константные данные**. Каждая из четырех комбинаций обеспечивает различные *привилегии доступа*. Подробнее о них рассказывается в следующих нескольких примерах.

7.5.1 Преобразование строк в верхний регистр с использованием изменяемого указателя на изменяемые данные

Наивысшие привилегии доступа к данным выдаются, когда функция получает **изменяемый указатель на изменяемые данные**. В этом случае

данные можно изменять посредством разыменования указателя, и сам указатель можно изменять с целью получить доступ к другим элементам данных. Объявление изменяемого указателя на изменяемые данные выполняется без использования квалификатора `const`. Такой указатель может использоваться, например, для передачи строки функции, которая обрабатывает (и возможно, изменяет) каждый символ в строке. В примере 7.4 представлена функция `convertToUppercase`, объявляющая свой параметр `sPtr` (`char *sPtr`) в строке 19 как *изменяемый указатель на изменяемые данные*. Функция обрабатывает массив `string` (на который указывает аргумент `sPtr`) по одному символу. Она преобразует каждый символ в верхний регистр, вызывая стандартную функцию `toupper` (строка 22) из заголовочного файла `<ctype.h>`, – если оригинальный символ не является буквой или уже является буквой верхнего регистра, `toupper` вернет оригинальный символ. Инструкция в строке 23 перемещает указатель к следующему символу.

Пример 7.4 | Преобразование символов строки в верхний регистр

```
 1 // Пример 7.4: fig07_10.c
 2 // Преобразование символов строки в верхний регистр с использованием
 3 // изменяемого указателя на изменяемые данные.
 4 #include <stdio.h>
 5 #include <ctype.h>
 6
 7 void convertToUppercase( char *sPtr ); // прототип
 8
 9 int main( void )
10 {
11    char string[] = "cHaRaCters and $32.98"; // инициализировать массив
12
13    printf( "The string before conversion is: %s", string );
14    convertToUppercase( string );
15    printf( "\nThe string after conversion is: %s\n", string );
16 } // конец main
17
18 // преобразует символы в строке в верхний регистр
19 void convertToUppercase( char *sPtr )
20 {
21    while( *sPtr != '\0' ){ // текущий символ не равен '\0'
22       *sPtr = toupper( *sPtr ); // преобразовать в верхний регистр
23       ++sPtr; // переустановить sPtr на следующий символ
24    } // конец while
25 } // конец функции convertToUppercase
```

```
The string before conversion is: cHaRaCters and $32.98
The string after conversion is: CHARACTERS AND $32.98
```

7.5.2 Вывод строки по одному символу с использованием изменяемого указателя на константные данные

Изменяемый указатель на константные данные *можно изменить*, сохранив в нем адрес любого другого элемента данных соответствующего типа,

но сами *данные*, на которые он указывает, *изменить нельзя*. Такой указатель можно использовать для приема аргумента-массива в функции, обрабатывающей каждый элемент массива, но не изменяющей данные. Например, функция `printCharacters` (пример 7.5) объявляет параметр `sPtr` типа `const char *` (строка 21). Объявление читается *справа налево* как «sPtr – это указатель на символьную константу». Функция с помощью инструкции `for` выводит символы строки, пока не встретит нулевого символа. После вывода каждого символа указатель `sPtr` передвигается на следующий символ в строке.

Пример 7.5 | Вывод строки по одному символу с использованием изменяемого указателя на константные данные

```
1 // Пример 7.5: fig07_11.c
2 // Вывод строки по одному символу с использованием
3 // изменяемого указателя на константные данные.
4
5 #include <stdio.h>
6
7 void printCharacters( const char *sPtr );
8
9 int main( void )
10 {
11    // инициализировать массив символов
12    char string[] = "print characters of a string";
13
14    puts( "The string is:" );
15    printCharacters( string );
16    puts( "" );
17 } // конец main
18
19 // указатель sPtr не может использоваться для изменения символа,
20 // на который он указывает, то есть sPtr – указатель "только для чтения"
21 void printCharacters( const char *sPtr )
22 {
23    // цикл по символам в строке
24    for ( ; *sPtr != '\0'; ++sPtr ) { // раздел инициализации отсутствует
25       printf( "%c", *sPtr );
26    } // конец for
27 } // конец функции printCharacters
```

```
The string is:
print characters of a string
```

Пример 7.12 демонстрирует функцию, принимающую изменяемый указатель (`xPtr`) на константные данные. В строке 18 функция пытается изменить данные, на которые указывает указатель `xPtr`, что вызывает ошибку компиляции. Фактический текст сообщения зависит от конкретного компилятора.

Пример 7.6 | Попытка изменить данные с помощью изменяемого указателя на константные данные

```
1 // Пример 7.6: fig07_12.c
2 // Попытка изменить данные с помощью
3 // изменяемого указателя на константные данные.
```

```
 4 #include <stdio.h>
 5 void f( const int *xPtr ); // прототип
 6
 7 int main( void )
 8 {
 9    int y; // определить переменную y
10
11    f( &y ); // f пытается выполнить запрещенное изменение
12 } // конец main
13
14 // указатель xPtr нельзя использовать для изменения
15 // значения переменной, на которую он указывает
16 void f( const int *xPtr )
17 {
18    *xPtr = 100; // ошибка: невозможно изменить константный объект
19 } // конец функции f
```

```
c:\examples\ch07\fig07_12.c(18) : error C2166: l-value specifies const object
```

Как известно, массивы относятся к составным типам данных, хранящим под одним именем взаимосвязанные элементы данных одного типа. В главе 10 мы познакомимся с еще одним представителем составных типов данных – **структурами** (иногда называемыми **записями** в других языках). Структура может хранить под одним именем взаимосвязанные элементы данных *разных* типов (например, информацию о каждом сотруднике компании). Когда в качестве аргумента функции передается массив, этот массив автоматически передается *по ссылке*. Но структуры всегда передаются *по значению* – функция получает *копию* структуры. На копирование элементов структуры тратится процессорное время. Поэтому, когда требуется передать структуру данных, для экономии времени мы можем передать функции указатель на константные данные и получить скорость передачи по ссылке и защищенность передачи по значению. При передаче указателя на структуру копируется только *адрес* структуры. В системе с 4-байтными адресами в этом случае достаточно скопировать всего 4 байта вместо всей структуры, размер которой может быть несравнимо больше.

Передача больших объектов, таких как структуры, с использованием указателей на константные данные позволяет получить преимущества производительности передачи по ссылке и защищенность передачи по значению.

При недостатке памяти и высоких требованиях к быстродействию используйте указатели. Если памяти достаточно и не требуется обеспечить максимальное быстродействие, передавайте данные по значению, проводя в жизнь принцип наименьших привилегий. Имейте в виду, что некоторые системы поддерживают квалификатор типа `const` недостаточно хорошо, поэтому передача по значению остается лучшим способом предотвратить изменение данных.

7.5.3 Попытка изменить константный указатель на изменяемые данные

Константный указатель на изменяемые данные всегда ссылается на *один и тот же* адрес в памяти, и данные, находящиеся там, *можно изменить* посредством этого указателя. Примером такого указателя может служить имя массива. Имя массива постоянно указывает на его начало. Все данные в массиве доступны и могут изменяться с использованием имени этого массива и индексов. Константный указатель на изменяемые данные можно использовать в функциях для приема аргументов-массивов, обеспечивающих доступ к элементам массивов посредством индексов. Указатели, объявленные с квалификатором `const`, должны инициализироваться в инструкции определения (если указатель является параметром функции, он инициализируется указателем, передаваемым в вызов функции). Пример 7.7 реализует попытку изменить константный указатель. Указатель `ptr` типа `int * const` объявляется в строке 12. Это определение можно прочитать справа налево как «`ptr` – это константный указатель на целое число». Указатель инициализируется (строка 12) адресом целочисленной переменной `x`. Программа пытается присвоить указателю `ptr` адрес переменной `y` (строка 15), что вызывает ошибку компиляции.

Пример 7.7 | Попытка изменить константный указатель на изменяемые данные

```
 1 // Пример 7.7: fig07_13.c
 2 // Попытка изменить константный указатель на изменяемые данные.
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    int x; // определить переменную x
 8    int y; // определить переменную y
 9
10    // ptr - константный указатель на целое число, которое можно изменить
11    // посредством ptr, но ptr всегда указывает на один и тот же адрес
12    int * const ptr = &x;
13
14    *ptr = 7; // допустимо: *ptr не является константой
15    ptr = &y; // ошибка: ptr - константа; нельзя присвоить новый адрес
16 } // конец main
```

```
c:\examples\ch07\fig07_13.c(15) : error C2166: l-value specifies const object
```

7.5.4 Попытка изменить константный указатель на константные данные

Меньше всего привилегий доступа к данным дает **константный указатель на константные данные**. Такой указатель всегда ссылается на *один и тот же* адрес в памяти и *не* позволяет с его помощью *изменить* данные,

хранящиеся по этому адресу. Именно так должны передаваться массивы функциям, которые *только просматривают* их содержимое с применением индексов. Программа в примере 7.8 определяет указатель `ptr` (строка 13) типа `const int *const`. Это определение читается *справа налево* как «`ptr` – константный указатель на целочисленную константу». В примере показаны сообщения об ошибках, сгенерированные компилятором для строк в программе, где производится попытка изменить данные, на которые указывает `ptr` (строка 16), и изменить адрес, хранящийся в переменной-указателе (строка 17).

Пример 7.8 | Попытка изменить константный указатель на константные данные

```
1 // Пример 7.8: fig07_14.c
2 // Попытка изменить константный указатель на константные данные.
3 #include <stdio.h>
4
5 int main( void )
6 {
7    int x = 5; // инициализировать переменную x
8    int y;     // определить переменную y
9
10   // ptr - константный указатель на целочисленную константу. ptr всегда
11   // указывает на один и тот же адрес; целое число по этому адресу
12   // не может быть изменено
13   const int *const ptr = &x; // инициализация выполняется
14
15   printf( "%d\n", *ptr );
16   *ptr = 7; // ошибка: *ptr - константа; нельзя присвоить новое значение
17   ptr = &y; // ошибка: ptr - константа; нельзя присвоить новый адрес
18 } // конец main
```

```
c:\examples\ch07\fig07_14.c(16) : error C2166: l-value specifies const object
c:\examples\ch07\fig07_14.c(17) : error C2166: l-value specifies const object
```

7.6 Пузырьковая сортировка с передачей аргументов по ссылке

Попробуем улучшить программу пузырьковой сортировки из примера 6.13, задействовав в ней две функции – `bubbleSort` и `swap`. Функция `bubbleSort` сортирует массив. Для обмена местами элементов массива `array[j]` и `array[j + 1]` (пример 7.9) она вызывает функцию `swap` (строка 50). Не забывайте, что в языке C функции *скрывают информацию* друг от друга, поэтому `swap` не имеет доступа к отдельным элементам массива, обрабатываемого функцией `bubbleSort`. Так как функции `bubbleSort` *требуется*, чтобы функция `swap` имела доступ к элементам массива, которые меняются местами, `bubbleSort` передает оба эти элемента *по ссылке*, то есть явно передает адреса обоих элементов. Несмотря на то что при передаче массива целиком он

автоматически передается по ссылке, отдельные элементы массива и *скаляры* по умолчанию передаются по значению. Именно поэтому в `bubbleSort` используется оператор взятия адреса (&) для каждого элемента массива в вызове функции `swap` (строка 50):

```
swap( &array[ j ], &array[ j + 1 ] );
```

Функция `swap` принимает аргумент `&array[j]` в виде переменной-указателя `element1Ptr` (строка 58). Даже несмотря на то что имя `array[j]` неизвестно функции `swap`, тем не менее она может использовать имя `*element1Ptr` как синоним для `array[j]` – обращаясь к `*element1Ptr`, функция `swap` *фактически* обращается к `array[j]` в `bubbleSort`. Аналогично когда `swap` обращается к `*element2Ptr`, она *фактически* обращается к `array[j + 1]` в `bubbleSort`. Даже при том, что в функции `swap` нельзя записать:

```
int hold = array[ j ];
array[ j ] = array[ j + 1 ];
array[ j + 1 ] = hold;
```

точно такой же эффект достигается строками 60–62:

```
int hold = *element1Ptr;
*element1Ptr = *element2Ptr;
*element2Ptr = hold;
```

Пример 7.9 | Ввод значений в массив, сортировка в порядке возрастания и вывод получившегося массива

```
1 // Пример 7.9: fig07_15.c
2 // Ввод значений в массив, сортировка в порядке
3 // возрастания и вывод получившегося массива.
4 #include <stdio.h>
5 #define SIZE 10
6
7 void bubbleSort( int * const array, size_t size ); // прототип
8
9 int main( void )
10 {
11    // инициализировать массив a
12    int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };
13
14    size_t i; // счетчик
15
16    puts( "Data items in original order" );
17
18    // обойти элементы массива a
19    for ( i = 0; i < SIZE; ++i ) {
20       printf( "%4d", a[ i ] );
21    } // конец for
22
23    bubbleSort( a, SIZE ); // отсортировать массив
24
25    puts( "\nData items in ascending order" );
26
```

```
27    // обойти элементы массива a
28    for ( i = 0; i < SIZE; ++i ) {
29       printf( "%4d", a[ i ] );
30    } // конец for
31
32    puts( "" );
33 } // конец main
34
35 // сортирует массив чисел с применением алгоритма пузырьковой сортировки
36 void bubbleSort( int * const array, size_t size )
37 {
38    void swap( int *element1Ptr, int *element2Ptr ); // прототип
39    unsigned int pass; // счетчик проходов
40    size_t j;          // счетчик сравнений
41
42    // цикл, управляемый счетчиком проходов
43    for ( pass = 0; pass < size - 1; ++pass ) {
44
45       // цикл, управляемый счетчиком сравнений в каждом проходе
46       for ( j = 0; j < size - 1; ++j ) {
47
48          // поменять местами соседние элемменты, если необходимо
49          if ( array[ j ] > array[ j + 1 ] ) {
50             swap( &array[ j ], &array[ j + 1 ] );
51          } // конец if
52       } // конец внутреннего цикла for
53    } // конец внешнего цикла for
54 } // конец функции bubbleSort
55
56 // меняет местами значения в памяти, адресуемые указателями
57 // element1Ptr и element2Ptr
58 void swap( int *element1Ptr, int *element2Ptr )
59 {
60    int hold = *element1Ptr;
61    *element1Ptr = *element2Ptr;
62    *element2Ptr = hold;
63 } // конец функции swap
```

```
Data items in original order
   2    6    4    8   10   12   89   68   45   37

Data items in ascending order
   2    4    6    8   10   12   37   45   68   89
```

Следует также отметить некоторые особенности функции `bubbleSort`. Заголовок функции (строка 36) объявляет массив как `int * const array` вместо `int array[]`, показывая тем самым, что `bubbleSort` принимает одномерный массив (напомню, что эти две формы записи являются взаимозаменяемыми). Параметр `size` объявлен с квалификатором `const`, согласно принципу наименьших привилегий. Несмотря на то что в параметре `size` передается копия значения переменной в `main` и изменение копии не повлияет на фактическое значение переменной в `main`, тем не менее функции `bubbleSort` *не* требуется изменять `size` для решения ее задачи. Размер массива остается по-

стоянным в ходе выполнения `bubbleSort`. Поэтому параметр `size` объявлен с квалификатором `const`, чтобы гарантировать его неизменяемость.

Прототип функции `swap` (строка 38) включен в тело функции `bubbleSort`. Сделано это по той простой причине, что `bubbleSort` является единственной функцией, вызывающей `swap`. Размещение прототипа в `bubbleSort` ограничивает возможность вызова `swap` рамками `bubbleSort`. Если другие функции в программе, не имеющие доступа к прототипу `swap`, попытаются вызвать ее, компилятор сгенерирует неизвестный прототип автоматически. Обычно такой автоматически сгенерированный прототип не совпадает с заголовком функции (что вызывает ошибку или предупреждение на этапе компиляции), потому что компилятор предполагает, что функция должна принимать и возвращать значение типа `int`.

Размещение прототипа в определении другой функции способствует продвижению принципа наименьших привилегий, ограничивая возможность вызова функции только там, где присутствует ее прототип.

Функция `bubbleSort` принимает размер массива в виде параметра (строка 36). Функция должна знать размер массива, чтобы отсортировать его. Когда функции передается массив, она получает адрес первого элемента в памяти. Разумеется, по адресу *нельзя* определить количество элементов в массиве. Поэтому необходимо также передать размер массива. Часто в практике используется другой прием, когда функции передается указатель на начало массива и указатель на первую ячейку памяти непосредственно за последним элементом массива – как будет показано в разделе 7.8, разность двух указателей дает длину массива, а получающийся код выглядит проще.

В примере 7.9 размер массива явно передается функции `bubbleSort`. Такой подход имеет еще два преимущества – возможность *повторного использования кода и надежность программной архитектуры*. Определяя функцию, принимающую размер массива в аргументе, мы обеспечиваем возможность ее использования в любых программах, оперирующих одномерными массивами любого размера.

Передавая функции массив, передавайте также его размер. Это поможет упростить использование данной функции в других программах.

Мы могли бы хранить размер сортируемого массива в глобальной переменной, доступной из любой точки в программе. Это немного повысило бы эффективность за счет отсутствия операции копирования размера и передачи его функции. Однако в других программах, где требуется возможность сортировки массивов целых чисел, подобная глобальная переменная может отсутствовать, и поэтому они не смогут использовать данную функцию.

Применение глобальных переменных часто ведет к нарушению принципа наименьших привилегий и снижению надежности программ. Глобальные переменные следует использовать только для представления по-настоящему разделяемых ресурсов, таких как текущее время.

Размер массива можно было бы также «зашить» непосредственно в функцию. Однако подобный прием сделал бы невозможным применение функции к массивам других размеров и существенно ограничил бы область ее использования. Такую функцию можно было бы использовать лишь в программах, обрабатывающих одномерные массивы того же размера.

7.7 Оператор sizeof

В языке C имеется специальный унарный оператор `sizeof`, возвращающий размер в байтах массива (или элемента данных любого другого типа). Когда этот оператор применяется к имени массива, как в примере 7.10 (строка 15), он возвращает общее количество байтов в массиве в виде значения типа `size_t`. Значения типа `float` на компьютере, где мы выполняли тестирование, занимают в памяти по 4 байта. Массив содержит 20 элементов этого типа. Поэтому общий размер массива составляет 80 байт.

Оператор `sizeof` вычисляет свой результат на этапе компиляции, поэтому он не несет никаких накладных расходов во время выполнения.

Пример 7.10 | Применение оператора sizeof к имени массива, возвращающего размер массива в байтах

```
 1 // Пример 7.10: fig07_16.c
 2 // Применение оператора sizeof к имени массива,
 3 // возвращающего размер массива в байтах.
 4 #include <stdio.h>
 5 #define SIZE 20
 6
 7 size_t getSize( float *ptr ); // прототип
 8
 9 int main( void )
10 {
11    float array[ SIZE ]; // создать массив
12
13    printf( "The number of bytes in the array is %u"
14            "\nThe number of bytes returned by getSize is %u\n",
15            sizeof( array ), getSize( array ) );
16 } // конец main
17
18 // возвращает размер ptr
19 size_t getSize( float *ptr )
20 {
21    return sizeof( ptr );
22 } // конец функции getSize
```

```
The number of bytes in the array is 80
The number of bytes returned by getSize is 4
```

Количество элементов в массиве тоже можно определить с помощью оператора `sizeof`. Например, взгляните на следующее определение массива:

```
double real[ 22 ];
```

Значения типа `double` обычно занимают 8 байт памяти. Отсюда несложно вычислить, что массив `real` занимает 176 байт. Чтобы определить количество элементов в массиве, можно использовать следующее выражение:

```
sizeof( real ) / sizeof( real[ 0 ] )
```

Это выражение определяет количество байтов, занимаемых массивом `real`, и делит его на количество байтов, занимаемых первым элементом этого массива (значением типа `double`).

Даже при том, что функция `getSize` принимает массив из 20 элементов, ее параметр `ptr` является обычным указателем на первый элемент массива. Когда оператор `sizeof` применяется к указателю, он возвращает *размер указателя*, а не размер объекта в памяти, на который тот указывает. В нашей системе размер указателя равен 4 байтам, поэтому `getSize` возвращает 4. Кроме того, прием определения количества элементов в массиве с помощью оператора `sizeof` можно использовать, *только* когда оператору `sizeof` передается фактический массив, а *не* указатель на него.

Определение размеров стандартных типов, массивов и указателей

Программа в примере 7.11 вычисляет количество байтов, занимаемых значениями основных стандартных типов. Результат работы этой программы зависит от реализации и иногда даже от применяемого компилятора.

Пример 7.11 | Использование оператора sizeof для определения размеров значений стандартных типов

```
// Пример 7.11: fig07_17.c
// Использование оператора sizeof для определения
// размеров значений стандартных типов
#include <stdio.h>

int main( void )
{
   char c;
   short s;
   int i;
   long l;
   long long ll;
   float f;
   double d;
   long double ld;
   int array[ 20 ];  // создать массив с 20 элементами типа int
   int *ptr = array; // создать указатель на массив
```

```
18
19    printf( "    sizeof c = %u\tsizeof(char) = %u"
20            "\n   sizeof s = %u\tsizeof(short) = %u"
21            "\n   sizeof i = %u\tsizeof(int) = %u"
22            "\n   sizeof l = %u\tsizeof(long) = %u"
23            "\n   sizeof ll = %u\tsizeof(long long) = %u"
24            "\n   sizeof f = %u\tsizeof(float) = %u"
25            "\n   sizeof d = %u\tsizeof(double) = %u"
26            "\n   sizeof ld = %u\tsizeof(long double) = %u"
27            "\n   sizeof array = %u"
28            "\n   sizeof ptr = %u\n",
29           sizeof c, sizeof( char ), sizeof s, sizeof( short ), sizeof i,
30           sizeof( int ), sizeof l, sizeof( long ), sizeof ll,
31           sizeof( long long ), sizeof f, sizeof( float ), sizeof d,
32           sizeof( double ), sizeof ld, sizeof( long double ),
33           sizeof array, sizeof ptr );
34 } // конец main
```

```
    sizeof c = 1       sizeof(char)  = 1
    sizeof s = 2       sizeof(short) = 2
    sizeof i = 4       sizeof(int) = 4
    sizeof l = 4       sizeof(long) = 4
   sizeof ll = 8       sizeof(long long) = 8
    sizeof f = 4       sizeof(float) = 4
    sizeof d = 8       sizeof(double) = 8
   sizeof ld = 8       sizeof(long double) = 8
sizeof array = 80
  sizeof ptr = 4
```

Количество байтов, занимаемых значением определенного типа, может отличаться в разных системах. Если вы пишете программы, правильность работы которых зависит от размеров типов данных и которые должны работать в разных операционных системах, используйте оператор `sizeof` *для определения количества байтов, занимаемых значениями.*

Оператор `sizeof` может применяться к именам любых переменных, типам или значениям (включая значения выражений). Когда этот оператор применяется к имени переменной (то есть *не* к имени массива) или константы, он возвращает количество байтов, занимаемых этой переменной или значением, представляемым константой. Когда операндом оператора `sizeof` является тип данных, его следует заключать в круглые скобки.

7.8 Выражения с указателями и арифметика указателей

Указатели являются допустимыми операндами в арифметических выражениях, выражениях присваивания и сравнения. Однако далеко не все операторы, обычно используемые в выражениях, могут использоваться в выражениях с указателями. В этом разделе описываются операторы, которые могут

принимать указатели в качестве операндов, и порядок использования этих операторов.

Для выполнения операций с указателями допускается использовать лишь ограниченное подмножество арифметических операторов. Указатели могут увеличиваться с помощью оператора *инкремента* (`++`) или уменьшаться с помощью оператора *декремента* (`--`), к указателям можно *прибавлять* целые числа (`+` и `+=`), из указателей можно *вычитать* целые числа (`-` и `-=`), и один указатель может вычитаться из другого – последняя операция имеет смысл, только если *оба* указателя указывают на элементы *одного и того же* массива.

Допустим, что определен массив `int v[5]`, а его первый элемент хранится в памяти по адресу `3000`. Предположим, что указатель `vPtr` инициализирован адресом первого элемента `v[0]`, то есть `vPtr` имеет значение `3000`. Эта ситуация, как она имеет место быть на компьютере с 4-байтными целыми числами, изображена на рис. 7.6. Переменная `vPtr` может быть инициализирована указателем на массив `v` одной из следующих инструкций:

```
vPtr = v;
vPtr = &v[ 0 ];
```

Так как результаты арифметической операции с указателем зависят от размера объекта, на который ссылается указатель, арифметика указателей в значительной степени зависит от аппаратной и программной архитектуры компьютера.

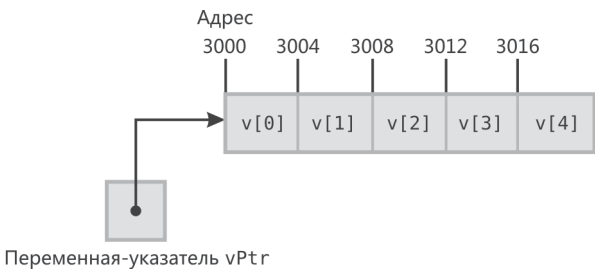


Рис. 7.6 | Массив v и переменная-указатель vPtr, ссылающаяся на v

В обычной арифметике выражение `3000 + 2` дает в результате `3002`. Но в арифметике указателей это не всегда так. Когда целое число прибавляется к указателю или вычитается из него, указатель не просто увеличивается или уменьшается на это число, а на данное число, умноженное на размер объекта, на который ссылается указатель. Размер объекта зависит от его типа. Например, инструкция

```
vPtr += 2;
```

присвоит переменной `vPtr` адрес `3008` (`3000 + 2 * 4`), если исходить из предположения, что значение типа `int` занимает 4 байта. Теперь `vPtr` будет указывать на элемент массива `v[2]` (рис. 7.7). В системе, где значение типа `int` занимает 2 байта, предыдущая инструкция присвоила бы переменной `vPtr` адрес `3004` (`3000 + 2 * 2`). Если бы массив `v` хранил элементы другого типа, предыдущая инструкция увеличила бы адрес в переменной `vPtr` на количество байтов, занимаемое этим типом, умноженное на два. При выполнении арифметических операций с указателями на массивы символов результаты будут соответствовать обычным арифметическим операциям, потому что один символ занимает 1 байт.

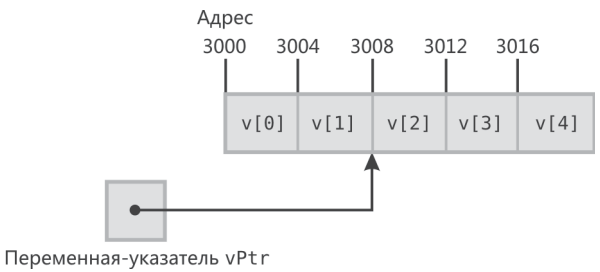


Рис. 7.7 | Указатель vPtr после арифметической операции

Если в какой-то момент времени указатель `vPtr` был увеличен до значения `3016`, что соответствует адресу элемента `v[4]`, тогда инструкция

```
vPtr -= 4;
```

уменьшит его до значения `3000` – адреса начала массива. Для увеличения или уменьшения указателя на единицу можно использовать операторы инкремента (`++`) и декремента (`--`). Любая из следующих инструкций

```
++vPtr;
vPtr++;
```

увеличит указатель, передвинув его на *следующий* элемент массива. Любая из следующих инструкций

```
--vPtr;
vPtr--;
```

уменьшит указатель, передвинув его на *предыдущий* элемент массива.

Переменные-указатели могут вычитаться друг из друга. Например, если предположить, что `vPtr` хранит адрес `3000`, а `v2Ptr` – адрес 3008, тогда инструкция

```
x = v2Ptr - vPtr;
```

присвоит переменной `x` *количество элементов массива* от `vPtr` до `v2Ptr`, в данном случае 2 (не 8). Арифметика указателей не определена, если указатели не ссылаются на массив. Мы не можем предполагать, что две переменные одного типа хранятся в памяти по соседству, если только они не являются соседними элементами массива.

Применять арифметику указателей к указателю, который ссылается не на элемент массива, обычно является ошибкой.

Вычитать или сравнивать два указателя, которые ссылаются не на элементы одного массива, обычно является ошибкой.

Выходить за пределы массива, используя арифметику указателей, является ошибкой.

Значение переменной-указателя может быть присвоено другой переменной-указателю, если обе переменные одного и того же типа. Исключением из этого правила является указатель на значение типа `void` (то есть `void *`), который считается *универсальным указателем*, способным представлять указатель любого типа. Указателю типа `void *` можно присвоить указатель любого типа, и указатель типа `void *` можно присвоить указателю любого типа. Ни в одном из этих случаев не требуется выполнять операцию приведения типа.

Указатель типа `void *` *не может* быть разыменован. Представьте такую ситуацию: компилятор знает, что указатель типа `int *` ссылается на 4-байтный объект в системе, где значение типа `int` занимает 4 байта, а указатель типа `void *` просто содержит адрес значения *неизвестного* типа – точное количество байтов, занимаемое объектом, на который ссылается указатель, компилятору *неизвестно*. Однако, чтобы разыменовать указатель, компилятор *должен* знать, сколько байтов занимает объект, адресуемый указателем.

Попытка присвоить значение указателя одного типа указателю другого типа, если ни один из них не является указателем типа `void *`, вызовет синтаксическую ошибку.

Попытка разыменовать указатель типа `void *` вызовет синтаксическую ошибку.

Указатели могут сравниваться с помощью операторов сравнения и операторов отношения, но такое сравнение бессмысленно, если указатели не ссылаются на элементы *одного* массива. При сравнении указателей сравниваются адреса в памяти. Сравнение двух указателей, ссылающихся на элементы одного массива, может помочь определить, например, какой из них ссылается на элемент с большим индексом. Чаще всего операция сравнения с указателями используется, чтобы определить, не содержит ли указатель значения `NULL`.

7.9 Связь между указателями и массивами

Массивы и указатели в языке C имеют очень тесную связь и часто могут использоваться взаимозаменяемо. Имя массива можно рассматривать как *константный указатель*. Указатели можно использовать для выполнения любых операций, вовлекающих индексы.

Допустим, что в программе определены массив целых чисел `b[5]` и указатель на целое число `bPtr`. Так как имя массива (без индекса) является указателем на первый элемент, мы можем присвоить переменной `bPtr` адрес первого элемента в массиве `b` инструкцией

```
bPtr = b;
```

Эта инструкция эквивалентна операции взятия адреса первого элемента массива:

```
bPtr = &b[ 0 ];
```

При желании доступ к элементу `b[3]` можно получить с помощью выражения

```
*( bPtr + 3 )
```

Число `3` в данном выражении – это **смещение** относительно указателя. Если указатель ссылается на первый элемент массива, смещение определяет адрес элемента того же массива, возвращаемый выражением, и в этом случае значение смещения эквивалентно индексу элемента в массиве. Такую форму записи часто называют **указатель/смещение**. Круглые скобки необходимы лишь по той простой причине, что оператор * имеет более высокий приоритет, чем оператор +. Без круглых скобок выражение выше прибавило бы число `3` к значению выражения `*bPtr` (то есть к значению элемента `b[0]`, если исходить из предположения, что переменная `bPtr` указывает на первый элемент массива). Выражение взятия адреса элемента массива

```
&b[ 3 ]
```

можно также записать как

```
bPtr + 3
```

Имя самого массива тоже можно интерпретировать как указатель и применять к нему арифметику указателей. Например, выражение

```
*( b + 3 )
```

также вернет значение элемента `b[ 3 ]`. Вообще говоря, любые выражения с участием индексов элементов массивов можно записать с помощью указателей и смещений. В данном случае форма записи указатель/смещение была применена к имени массива. Предыдущая инструкция не изменяет значения имени массива; `b` по-прежнему указывает на первый элемент.

Указатели можно снабжать индексами, подобно именам массивов. Если предположить, что `bPtr` хранит адрес `b`, выражение

```
bPtr[ 1 ]
```

будет ссылаться на элемент `b[ 1 ]`. Эту форму записи часто называют **указатель/индекс**.

Помните, что имя массива фактически является константным указателем; оно всегда указывает на начало массива. То есть выражение

```
b += 3
```

недопустимо, потому что пытается изменить значение имени массива с применением арифметики указателей.

Попытка изменить значение имени массива с помощью арифметики указателей вызывает ошибку компиляции.

В примере 7.12 демонстрируются четыре способа обращения к элементам массива, обсуждавшиеся выше, – с применением индекса к имени массива, с использованием формы записи указатель/смещение с именем массива, с применением индекса к указателю и с использованием формы записи указатель/смещение с указателем – для вывода четырех элементов массива `b`.

Пример 7.12 | Использование индексов и указателей для обращения к элементам массива

```
 1 // Пример 7.12: fig07_20.cpp
 2 // Использование индексов и указателей для обращения к элементам массива.
 3 #include <stdio.h>
 4 #define ARRAY_SIZE 4
 5
 6 int main( void )
 7 {
 8    int b[] = { 10, 20, 30, 40 }; // создать и инициализировать масив b
 9    int *bPtr = b; // создать указатель и присвоить ему адрес массива b
10    size_t i;      // счетчик
11    size_t offset; // счетчик
12
13    // вывести массив b с использованием индексов
```

```
14    puts( "Array b printed with:\nArray subscript notation" );
15
16    // цикл по элементам массива b
17    for ( i = 0; i < ARRAY_SIZE; ++i ) {
18       printf( "b[ %u ] = %d\n", i, b[ i ]);
19    } // конец for
21    // вывести массив b с использованием формы записи указатель/смещение
22    puts( "\nPointer/offset notation where\n"
23          "the pointer is the array name" );
24
25    // цикл по элементам массива b
26    for ( offset = 0; offset < ARRAY_SIZE; ++offset ) {
27       printf( "*( b + %u ) = %d\n", offset, *( b + offset ) );
28    } // конец for
29
30    // вывести массив b с использованием bPtr и индексов
31    puts( "\nPointer subscript notation" );
32
33    // цикл по элементам массива b
34    for ( i = 0; i < ARRAY_SIZE; ++i ) {
35       printf( "bPtr[ %u ] = %d\n", i, bPtr[ i ]);
36    } // конец for
37
38    // вывести массив с использованием bPtr и формы записи
      // указатель/смещение
39    puts( "\nPointer/offset notation" );
40
41    // цикл по элементам массива b
42    for ( offset = 0; offset < ARRAY_SIZE; ++offset ) {
43       printf( "*( bPtr + %u ) = %d\n", offset, *( bPtr + offset ) );
44    } // конец for
45 } // конец main
```

```
Array b printed with:
Array subscript notation
b[ 0 ] = 10
b[ 1 ] = 20
b[ 2 ] = 30
b[ 3 ] = 40

Pointer/offset notation where
the pointer is the array name
*( b + 0 ) = 10
*( b + 1 ) = 20
*( b + 2 ) = 30
*( b + 3 ) = 40

Pointer subscript notation
bPtr[ 0 ] = 10
bPtr[ 1 ] = 20
bPtr[ 2 ] = 30
bPtr[ 3 ] = 40

Pointer/offset notation
*( bPtr + 0 ) = 10
*( bPtr + 1 ) = 20
*( bPtr + 2 ) = 30
*( bPtr + 3 ) = 40
```

Копирование строк с применением массивов и указателей

Для дальнейшей демонстрации взаимозаменяемости массивов и указателей рассмотрим две функции копирования строк – `copy1` и `copy2` – в программе из примера 7.13. Обе функции копируют строку в массив символов. Прототипы функций `copy1` и `copy2` выглядят идентичными. Они решают одну и ту же задачу, но реализованы по-разному.

Пример 7.13 | Копирование строки с использованием формы записи обращения к массиву и указателю

```
// Пример 7.13: fig07_21.c
// Копирование строки с использованием массива и указателя.
#include <stdio.h>
#define SIZE 10

void copy1( char * const s1, const char * const s2 ); // прототип
void copy2( char *s1, const char *s2 );               // прототип

int main( void )
{
   char string1[ SIZE ];        // создать массив string1
   char *string2 = "Hello";     // создать указатель на строку
   char string3[ SIZE ];        // создать массив string3
   char string4[] = "Good Bye"; // создать указатель на строку

   copy1( string1, string2 );
   printf( "string1 = %s\n", string1 );

   copy2( string3, string4 );
   printf( "string3 = %s\n", string3 );
} // конец main

// копирует s2 в s1 с использованием формы записи обращения к массиву
void copy1( char * const s1, const char * const s2 )
{
   size_t i; // счетчик

   // цикл по символам в строках
   for ( i = 0; ( s1[ i ] = s2[ i ] ) != '\0'; ++i ) {
      ; // пустое тело цикла
   } // конец for
} // конец функции copy1

// копирует s2 в s1 с использованием формы записи обращения к указателю
void copy2( char *s1, const char *s2 )
{
   // цикл по символам в строках
   for ( ; ( *s1 = *s2 ) != '\0'; ++s1, ++s2 ) {
      ; // пустое тело цикла
   } // конец for
} // конец функции copy2
```

```
string1 = Hello
string3 = Good Bye
```

Для копирования строки `s2` в массив символов `s1` функция `copy1` использует *форму обращения к элементам массива*. Для перечисления индексов функция определяет переменную-счетчик `i`. Собственно операция копирования выполняется заголовком инструкции `for` (строка 29) – тело цикла составляет пустая инструкция. В заголовке этой инструкции `for` переменная `i` инициализируется нулем и увеличивается на 1 в каждой итерации цикла. Выражение `s1[i] = s2[i]` копирует один символ из `s2` в `s1`. Когда в `s2` встречается нулевой символ, он копируется в `s1` и результат операции присваивания становится значением левого операнда оператора сравнения. В результате условие продолжения цикла становится ложным и копирование прекращается.

Для копирования строки `s2` в массив символов `s1` функция `copy2` использует *форму обращения к указателю и арифметику указателей*. Опять же, копирование выполняется заголовком инструкции `for` (строка 38). На этот раз заголовок инструкции не выполняет инициализацию каких-либо переменных. Как и в функции `copy1`, копирование выполняется выражением `(*s1 = *s2)`. Это выражение разыменовывает указатель `s2` и полученный символ присваивает разыменованному указателю `*s1`. После присваивания в условном выражении указатели перемещаются операцией инкремента на следующий символ: `s1` – в массиве и `s2` – в строке соответственно. Когда в строке `s2` встречается нулевой символ, он присваивается разыменованному указателю `s1`, и цикл завершается.

В первом аргументе обеим функциям – `copy1` и `copy2` – должен передаваться массив достаточного размера, чтобы вместить строку, переданную во втором аргументе. В противном случае при попытке записи за пределы массива может произойти ошибка. Обратите также внимание, что второй параметр в обеих функциях объявлен как `const char *` (константная строка). Обе функции копируют второй аргумент в первый – они читают символы из второго аргумента по одному, но *не изменяют их*. То есть второй параметр объявлен как указатель на константные данные, исходя из *принципа наименьших привилегий* – ни одной из двух функций не требуется изменять второй аргумент, поэтому ни одной из них не дается такая возможность.

7.10 Массивы указателей

Массивы могут хранить не только обычные значения, но и указатели. Часто возможность создания **массивов указателей** используется для определения **массивов строк**. Каждый элемент такого массива является строкой, но в языке C строка фактически является указателем на первый символ этой строки. То есть каждый элемент массива строк в действительности является указателем. Взгляните на определение массива строк `suit`, который может пригодиться для представления колоды игральных карт:

```
const char *suit[4] = { "Hearts", "Diamonds", "Clubs", "Spades" };
```

Фрагмент `suit[4]` определения указывает, что массив содержит 4 элемента. Фрагмент `char *` сообщает, что каждый элемент массива `suit` имеет тип «указатель на символ». Квалификатор `const` говорит, что строки, на которые указывают элементы массива, не могут изменяться с их помощью. Массив инициализирован четырьмя элементами: `"Hearts"`, `"Diamonds"`, `"Clubs"` и `"Spades"`. Каждый хранится в памяти как *строка, завершающаяся нулевым символом*, на один символ длиннее, чем строки, заключенные в кавычки. Строки имеют длину 7, 9, 6 и 7 символов соответственно. На первый взгляд, кажется, что массив хранит фактические строки, однако в действительности его элементами являются указатели (рис. 7.8). Каждый указатель ссылается на первый символ соответствующей строки. То есть даже при том, что массив `suit` имеет *фиксированный* размер, он обеспечивает доступ к строкам *произвольной длины*. Такая гибкость является одним из примеров широких возможностей представления структур данных в языке C.

Ту же информацию можно было бы поместить в двумерный массив, в котором каждая строка представляет отдельную карточную масть, а каждый столбец – букву в названии масти. При такой организации структуры данных пришлось бы определить фиксированное количество столбцов, достаточное для хранения самой длинной из возможных строк. Подобный подход может приводить к напрасному расходованию памяти, когда нужно хранить большое количество строк, значительная часть которых существенно короче самой длинной строки. Массив с названиями мастей игральных карт мы будем использовать в следующем разделе.

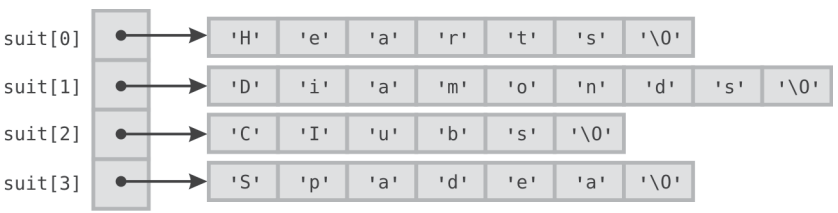


Рис. 7.8 | Графическое представление массива suit

7.11 Пример: тасование и раздача карт

Теперь мы воспользуемся генератором случайных чисел для реализации программы тасования и раздачи игральных карт. Код этой программы затем может использоваться для создания программ, реализующих карточные игры. Чтобы продемонстрировать некоторые малозаметные проблемы производительности, мы преднамеренно использовали не самый оптимальный алгоритм тасования и раздачи. В главе 10 мы представим вашему вниманию более эффективный алгоритм.

Используя нисходящий стиль разработки, мы создадим программу, которая будет тасовать колоду из 52 игральных карт и затем раздавать их все. Нисходящий стиль особенно удобно использовать для решения более крупных и сложных задач, чем мы показывали в предыдущих главах.

Для представления колоды игральных карт мы будем использовать двумерный массив 4 на 13 (рис. 7.9). Строки в этом массиве соответствуют *мастям*. Например, строка 0 соответствует масти «черви» (Hearts), строка 1 – «бубны» (Diamonds), строка 3 – «трефы» (Clubs), и строка 3 – «пики» (Spades). Колонки соответствуют *значениям* карт – колонки от 0 до 9 соответствуют значениям карт от туза (Ace) до десятки соответственно, а колонки от 10 до 12 – значениям карт «валет» (Jack), «дама» (Queen) и «король» (King). Мы также определим массив строк с названиями четырех мастей и массив строк с названиями тринадцати значений карт.

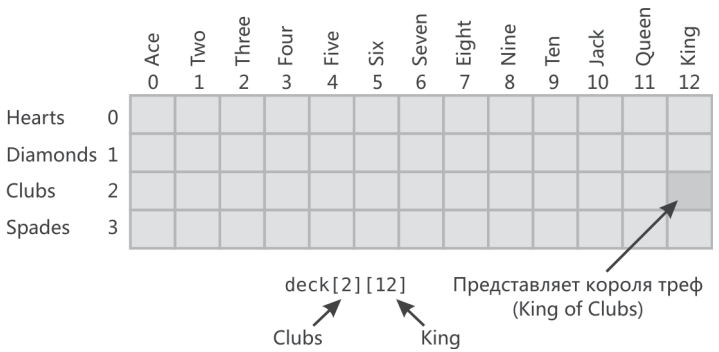


Рис. 7.9 | Двумерный массив, представляющий колоду карт

При таком представлении колоды карт сымитировать ее *перемешивание* можно, как описывается далее. Сначала массив с картами заполняется нулями. Затем выбираются два *случайных* числа: номер строки `row` (0–3) и номер столбца `column` (0–12). В соответствующий элемент массива `deck[row][column]` записывается число 1, указывающее, что данная карта будет выбрана первой при раздаче. Этот процесс продолжается, пока в массив не будут вставлены числа 2, 3, …, 52, соответствующие второй, третьей, ..., пятьдесят второй карте в колоде. По мере заполнения массива увеличивается вероятность случайного выбора номеров строки и столбца, то есть значений `row` и `column`, для которых элемент `deck[row][column]` уже имеет ненулевое значение. В этом случае полученные случайные числа отбрасываются, и попытки получить случайные значения повторяются, пока не будут получены индексы элемента с нулевым значением. В конечном итоге все 52 элемента массива будут заполнены числами от 1 до 52. В этот момент тасование колоды карт завершается.

Такой алгоритм тасования колоды может выполняться *до бесконечности*, если прежде выбранные карты будут выбираться снова и снова. Этот феномен называется **перенос завершения на неопределенное время** (indefinite postponement).

Иногда алгоритм, кажущийся «естественным», может содержать малозаметные проблемы производительности, такие как перенос завершения на неопределенное время. Старайтесь выбирать алгоритмы, исключающие вероятность подобного переноса.

Чтобы раздать первую карту, мы находим элемент массива `deck[row][column]` со значением 1. Поиск реализуется парой вложенных инструкций `for`, перебирающих номера строк от 0 до 3 и номера столбцов от 0 до 12. А какая карта соответствует этому элементу массива? Массив `suit` содержит названия четырех мастей карт, и чтобы получить масть, достаточно обратиться к строке `suit[row]`. Аналогично, чтобы получить значение карты, достаточно обратиться к строке `face[column]`. Дополнительно программа выводит строку `" of "`. Выводя эту информацию в надлежащем порядке, мы получим название каждой карты в виде `"King of Clubs"` (король треф) , `"Ace of Diamonds"` (туз бубен) и т. д.

Программа тасования и раздачи колоды карт представлена в примере 7.14, а результат ее выполнения – в примере 7.15. Для вывода строк с помощью функции `printf` используется спецификатор преобразования `%s`. Соответствующий ему аргумент в вызове `printf` должен быть указателем на значение типа `char` (или массив типа `char`). Спецификация формата `"%5s of %-8s"` (пример 7.14, строка 74) обеспечивает вывод строки из пяти символов с *выравниванием по правому краю*, за которой следуют символы `" of "` и строка из восьми символов с *выравниванием по левому краю*. Знак «минус» в спецификаторе `%-8s` обозначает выравнивание по левому краю.

Пример 7.14 | Тасование и раздача колоды игральных карт

```
 1 // Пример 7.14: fig07_24.c
 2 // Тасование и раздача колоды игральных карт.
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5 #include <time.h>
 6
 7 #define SUITS 4
 8 #define FACES 13
 9 #define CARDS 52
10
11 // прототипы
12 void shuffle( unsigned int wDeck[][ FACES ] ); // изменяет wDeck
13 void deal( unsigned int wDeck[][ FACES ], const char *wFace[],
14    const char *wSuit[] ); // раздача не изменяет массивы
15
16 int main( void )
17 {
```

```
18    // инициализировать массив с названиями мастей
19    const char *suit[ SUITS ] =
20       { "Hearts", "Diamonds", "Clubs", "Spades" };
21
22    // инициализировать массив с названиями значений карт
23    const char *face[ FACES ] =
24       { "Ace", "Deuce", "Three", "Four",
25         "Five", "Six", "Seven", "Eight",
26         "Nine", "Ten", "Jack", "Queen", "King" };
27
28    // инициализировать массив колоды карт
29    unsigned int deck[ SUITS ][ FACES ] = { 0 };
30
31    srand( time( NULL ) ); // инициализировать генератор случайных чисел
32
33    shuffle( deck ); // перетасовать колоду
34    deal( deck, face, suit ); // раздать колоду
35 } // конец main
36
37 // тасует колоду карт
38 void shuffle( unsigned int wDeck[][ FACES ] )
39 {
40    size_t row;    // номер строки
41    size_t column; // номер столбца
42    size_t card;   // счетчик
43
44    // выбрать случайные индексы карты в колоде
45    for ( card = 1; card <= CARDS; ++card ) {
46
47       // выбрать новые случайные индексы незанятого элемента
48       do {
49          row = rand() % SUITS;
50          column = rand() % FACES;
51       } while( wDeck[ row ][ column ] != 0 ); // конец do...while
52
53       // записать порядковый номер карты в колоде
54       wDeck[ row ][ column ] = card;
55    } // конец for
56 } // конец функции shuffle
57
58 // раздает колоду карт
59 void deal( unsigned int wDeck[][ FACES ], const char *wFace[],
60    const char *wSuit[] )
61 {
62    size_t card;   // счетчик карт
63    size_t row;    // счетчик строк
64    size_t column; // счетчик столбцов
65
66    // раздать карты
67    for ( card = 1; card <= CARDS; ++card ) {
68       // цикл по строкам в массиве wDeck
69       for ( row = 0; row < SUITS; ++row ) {
70          // цикл по столбцам в текущей строке массива wDeck
71          for ( column = 0; column < FACES; ++column ) {
72             // если значение элемента равно card, вывести карту
73             if ( wDeck[ row ][ column ] == card ) {
74                printf( "%5s of %-8s%c", wFace[ column ], wSuit[ row ],
```

```
75                   card % 2 == 0 ? '\n' : '\t' ); // вывод в 2 колонки
76             } // конец if
77          } // конец for
78       } // конец for
79    } // конец for
80 } // конец функции deal
```

Пример 7.15 | Пример выполнения программы раздачи карт

```
Nine of Hearts         Five of Clubs
Queen of Spades        Three of Spades
Queen of Hearts          Ace of Clubs
 King of Hearts          Six of Spades
 Jack of Diamonds       Five of Spades
Seven of Hearts         King of Clubs
Three of Clubs         Eight of Hearts
Three of Diamonds       Four of Diamonds
Queen of Diamonds       Five of Diamonds
  Six of Diamonds       Five of Hearts
  Ace of Spades          Six of Hearts
 Nine of Diamonds      Queen of Clubs
Eight of Spades         Nine of Clubs
Deuce of Clubs           Six of Clubs
Deuce of Spades         Jack of Clubs
 Four of Clubs         Eight of Clubs
 Four of Spades        Seven of Spades
Seven of Diamonds      Seven of Clubs
 King of Spades          Ten of Diamonds
 Jack of Hearts          Ace of Hearts
 Jack of Spades          Ten of Clubs
Eight of Diamonds      Deuce of Diamonds
  Ace of Diamonds       Nine of Spades
 Four of Hearts        Deuce of Hearts
 King of Diamonds        Ten of Spades
Three of Hearts          Ten of Hearts
```

Алгоритм раздачи карт имеет один недостаток. Даже если совпадение будет найдено, два вложенных цикла `for` продолжают просмотр оставшейся части колоды. Мы исправим этот недостаток в главе 10.

7.12 Указатели на функции

Указатель на функцию хранит *адрес* функции в памяти. В главе 6 мы видели, что имя массива в действительности является указателем на первый элемент этого массива. Аналогично имя функции в действительности является адресом первой инструкции в теле функции. Указатели на функции можно *передавать* другим функциям, *возвращать* из функций, *хранить* в массивах и *присваивать* другим указателям на функции.

Для иллюстрации использования указателей на функции в примере 7.16 приводится модифицированная версия программы пузырьковой сортировки из примера 7.9. Новая версия состоит из функции `main`, а также включа-

ет функции bubble, swap, ascending и descending. Помимо массива целых чисел и размера массива, функция bubble принимает также указатель на функцию – ascending или descending. Программа предлагает пользователю выбрать порядок сортировки массива *по возрастанию* или *по убыванию*. Если пользователь в ответ вводит число 1, в функцию bubble передается указатель на функцию ascending, в результате чего массив сортируется в порядке возрастания значений элементов. Если пользователь вводит 2, в функцию bubble передается указатель на функцию descending, и массив сортируется в порядке убывания. Вывод программы приводится в примере 7.17.

Пример 7.16 | Универсальная программа сортировки, использующая указатели на функции

```
// Пример 7.16: fig07_26.c
// Универсальная программа сортировки, использующая указатели на функции.
#include <stdio.h>
#define SIZE 10

// прототипы
void bubble( int work[], size_t size, int (*compare)( int a, int b ) );
int ascending( int a, int b );
int descending( int a, int b );

int main( void )
{
   int order;      // 1 - по возрастанию, 2 — по убыванию
   size_t counter; // счетчик

   // инициализировать неупорядоченный массив a
   int a[ SIZE ] = { 2, 6, 4, 8, 10, 12, 89, 68, 45, 37 };

   printf( "%s", "Enter 1 to sort in ascending order,\n"
                 "Enter 2 to sort in descending order: " );
   scanf( "%d", &order );

   puts( "\nData items in original order" );

   // вывести оригинальный массив
   for ( counter = 0; counter < SIZE; ++counter ) {
      printf( "%5d", a[ counter ] );
   } // конец for

   // сортировать массив по возрастанию; передать функцию ascending в
   // аргументе, чтобы обеспечить сортировку по возрастанию
   if ( order == 1 ) {
      bubble( a, SIZE, ascending );
      puts( "\nData items in ascending order" );
   } // конец if
   else { // передать функцию descending
      bubble( a, SIZE, descending );
      puts( "\nData items in descending order" );
   } // конец else
```

```
40
41    // вывести отсортированный массив
42    for ( counter = 0; counter < SIZE; ++counter ) {
43       printf( "%5d", a[ counter ] );
44    } // конец for
45
46    puts( "\n" );
47 } // конец main
48
49 // универсальная функция сортировки; параметр compare - это указатель на
50 // функцию сравнения, определяющую порядок сортировки
51 void bubble( int work[], size_t size, int (*compare)( int a, int b ) )
52 {
53    unsigned int pass; // счетчик проходов
54    size_t count;      // счетчик сравнений
55
56    void swap( int *element1Ptr, int *element2ptr ); // прототип
57
58    // цикл, управляющий количеством проходов
59    for ( pass = 1; pass < size; ++pass ) {
60
61       // цикл, управляющий количеством сравнений в одном проходе
62       for ( count = 0; count < size - 1; ++count ) {
63          // если соседние элементы стоят не в том порядке,
64          // поменять их местами
65          if ( (*compare)( work[ count ], work[ count + 1 ] ) ){
66             swap( &work[ count ], &work[ count + 1 ] );
67          } // конец if
68       } // конец for
69    } // конец for
70 } // конец функции bubble
71
72 // меняет местами значения элементов, на которые указывают
73 // element1Ptr и element2Ptr
74 void swap( int *element1Ptr, int *element2Ptr )
75 {
76    int hold; // временная переменная
77
78    hold = *element1Ptr;
79    *element1Ptr = *element2Ptr;
80    *element2Ptr = hold;
81 } // конец функции swap
82
83 // определяет, в правильном ли порядке стоят элементы при
84 // сортировке по возрастанию
85 int ascending( int a, int b )
86 {
87    return b < a; // следует поменять местами, если b < a
88 } // конец функции ascending
89
90 // определяет, в правильном ли порядке стоят элементы при
91 // сортировке по убыванию
92 int descending( int a, int b )
93 {
94    return b > a; // следует поменять местами, если b > a
95 } // конец функции descending
```

Пример 7.17 | Вывод программы сортировки из примера 7.16

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 1

Data items in original order
    2     6     4     8    10    12    89    68    45    37
Data items in ascending order
    2     4     6     8    10    12    37    45    68    89
```

```
Enter 1 to sort in ascending order,
Enter 2 to sort in descending order: 2

Data items in original order
    2     6     4     8    10    12    89    68    45    37
Data items in descending order
   89    68    45    37    12    10     8     6     4     2
```

В заголовке функции `bubble` (строка 51) объявлен следующий параметр:

```
int (*compare)( int a, int b )
```

Он сообщает, что параметр `(compare)` является указателем на функцию, которая имеет два целочисленных параметра и возвращает целочисленный результат. Круглые скобки вокруг `*compare` необходимы, чтобы сгруппировать оператор * с именем `compare` и тем самым показать, что `compare` является указателем. Если круглые скобки опустить, объявление пример вид:

```
int *compare( int a, int b )
```

что равносильно объявлению функции, принимающей два целых числа и возвращающей указатель на целое число.

В прототипе функции `bubble` (строка 7) определение третьего параметра имеет вид:

```
int (*)( int, int );
```

без имени-указателя на функцию и без имен параметров.

Функция, передаваемая функции `bubble`, вызывается в инструкции `if` (строка 65):

```
65          if ( (*compare)( work[ count ], work[ count + 1 ] ) ){
```

Так же как для доступа к значению переменной необходимо разыменовать указатель, *для вызова функции необходимо разыменовать указатель на функцию*.

Вызов функции можно произвести и без разыменования указателя, как, например, в инструкции:

```
if ( compare( work[ count ], work[ count + 1 ] ) )
```

где указатель используется непосредственно как имя функции. Мы предпочитаем использовать первый способ вызова функций по указателям, потому что он наглядно демонстрирует, что `compare` – это указатель на функцию, который разыменовывается в вызов функции. Второй способ делает имя указателя `compare` неотличимым от имени фактической функции, что может вызвать замешательство у тех, кто будет читать ваш код и, захотев посмотреть определение функции `compare`, обнаружит, что она *нигде не определена*.

Использование указателей на функции для создания программ, управляемых системой меню

Указатели на функции часто используются в текстовых программах, *управляемых системой меню*. Пользователю предлагается выбрать пункт меню (например, от 1 до 5) вводом номера элемента меню. Каждый пункт обслуживается отдельной функцией. Указатели на эти функции хранятся в массиве. Выбор пользователя интерпретируется как индекс элемента в массиве, используемого для вызова функции.

В примере 7.18 представлен обобщенный пример механизма определения и использования массива указателей на функции. В примере определяются три функции – `function1`, `function2` и `function3`, каждая из которых принимает целое число и ничего не возвращает. Указатели на эти функции сохраняются в массиве `f`, определяемом в строке 14.

Пример 7.18 | Демонстрация использования массива указателей на функции

```
1 // Пример 7.18: fig07_28.c
2 // Демонстрация использования массива указателей на функции.
3 #include <stdio.h>
4
5 // прототипы
6 void function1( int a );
7 void function2( int b );
8 void function3( int c );
9
10 int main( void )
11 {
12    // инициализировать массив с 3 указателями на функции, каждая из
13    // которых принимает целое число и ничего не возвращает
14    void (*f[ 3 ])( int ) = { function1, function2, function3 };
15
16    size_t choice; // переменная для хранения выбора пользователя
17
18    printf( "%s", "Enter a number between 0 and 2, 3 to end: " );
19    scanf( "%u", &choice );
20
21    // обработать выбор пользователя
22    while ( choice >= 0 && choice < 3 ) {
23
24       // вызвать функцию по указателю в элементе массива f
25       // с индексом choice и передать ей значение choice
26       (*f[ choice ])( choice );
```

```
27
28       printf( "%s", "Enter a number between 0 and 2, 3 to end: " );
29       scanf( "%u", &choice );
30    } // конец while
31
32    puts( "Program execution completed." );
33 } // конец main
34
35 void function1( int a )
36 {
37    printf( "You entered %d so function1 was called\n\n", a );
38 } // конец function1
39
40 void function2( int b )
41 {
42    printf( "You entered %d so function2 was called\n\n", b );
43 } // конец function2
44
45 void function3( int c )
46 {
47    printf( "You entered %d so function3 was called\n\n", c );
48 } // конец function3
```

```
Enter a number between 0 and 2, 3 to end: 0
You entered 0 so function1 was called

Enter a number between 0 and 2, 3 to end: 1
You entered 1 so function2 was called

Enter a number between 0 and 2, 3 to end: 2
You entered 2 so function3 was called

Enter a number between 0 and 2, 3 to end: 3
Program execution completed.
```

Определение массива `f` читается, начиная с самой левой пары скобок: «`f` – это массив с тремя указателями на функции, каждая из которых принимает целое число и ничего не возвращает». Массив инициализируется именами трех функций. Когда пользователь вводит значение в диапазоне от 0 до 2, введенное значение используется в качестве индекса в массиве указателей на функции. В вызове функции (строка 26) `f[choice]` извлекает указатель на функцию из элемента с индексом `choice`. Затем *указатель разыменовывается, чтобы вызвать функцию*, при этом функции передается значение `choice` в качестве аргумента. Каждая функция выводит значение своего аргумента и имя функции, чтобы показать, что вызов был выполнен правильно.

7.13 Безопасное программирование на C

printf_s, scanf_s и другие безопасные функции

В предыдущих разделах «Безопасное программирование на C» мы уже знакомились с функциями `printf_s` и `scanf_s`, и там же упоминались другие более безопасные версии функций из стандартной библиотеки, описывае-

мые приложением Annex K к стандарту языка C. Ключевой особенностью функций, таких как `printf_s` и `scanf_s`, делающей их более безопасными, является *требование времени выполнения* к аргументам-указателям, значения которых должны отличаться от `NULL`. Функции проверяют удовлетворение этих ограничений *перед* попыткой использовать указатели. Любой аргумент-указатель со значением `NULL` считается *нарушающим ограничения* и заставляет функцию вернуть признак ошибки. Если функция `scanf_s` получит какой-либо указатель (включая строку формата) со значением `NULL`, она вернет `EOF`. Если функция `printf_s` получит пустой указатель на строку формата или пустой указатель, соответствующий спецификатору `%s`, она прервет вывод данных и вернет отрицательное число. Более подробное описание функций, определяемых приложением Annex K, ищите в документации к своему компилятору.

Другие рекомендации CERT, касающиеся указателей

Неправильное использование указателей часто приводит к появлению уязвимостей в системах. Центр CERT предлагает различные рекомендации, следование которым может помочь предотвратить проблемы с безопасностью. Если вы занимаетесь созданием промышленных систем на языке C, обязательно ознакомьтесь с книгой «CERT C Secure Coding Standard», информацию о которой можно найти по адресу: `www.securecoding.cert.org`. Кроме того, приемы безопасного программирования с применением указателей описываются в следующих рекомендациях:

- EXP34-C: попытка разыменовать пустой (`NULL`) указатель обычно приводит к аварийному завершению программы, но сотрудникам CERT известны случаи, когда операции разыменования пустого указателя использовались злоумышленниками для выполнения произвольного кода;
- DCL13-C: в разделе 7.5 рассматривалась возможность применения квалификатора `const` к указателям. Если параметр функции указывает на значение, которое не должно изменяться этой функцией, следует использовать квалификатор `const`, чтобы указать, что данные не изменяются функцией. Например, чтобы объявить указатель на строку, которая не должна изменяться, следует использовать указатель типа `const char *`;
- MSC16-C: в этой рекомендации обсуждаются приемы шифрования указателей на функции с целью не дать злоумышленнику возможности затереть его и использовать для выполнения своего кода.

8

Символы и строки

В этой главе вы:

- познакомитесь с функциями для работы с символами (<ctype.h>);
- научитесь пользоваться функциями преобразований строк (<stdlib.h>);
- узнаете, как производить ввод/вывод строк и символов с помощью стандартной библиотеки ввода/вывода (<stdio.h>);
- увидите, как обрабатывать строки с применением функций из стандартной библиотеки (<string.h>);
- получите представление о строковых функциях для работы с памятью (<string.h>).

8.1 Введение

В этой главе будут представлены функции из стандартной библиотеки языка C, предназначенные для работы со строками и символами. Эти функции обеспечивают возможность работы с символами, последовательностями символов, строками текста и блоками памяти. В главе будут обсуждаться приемы, используемые при разработке редакторов, текстовых процессоров, программ постраничного просмотра, компьютерных систем верстки и других программных средств обработки текстовой информации. Операции с текстом, выполняемые функциями форматированного ввода/вывода, такими как `printf` и `scanf`, могут быть реализованы с применением функций, обсуждаемых в этой главе.

8.2 Основы работы со строками и символами

Символы являются основными строительными блоками, из которых составляются исходные тексты программ. Все программы конструируются из последовательностей символов, сгруппированных в осмысленные строки,

которые интерпретируются компьютерами как последовательности инструкций. Программа может содержать **символьные константы**. Символьная константа – это значение типа `int`, представленное символом в одиночных кавычках, то есть значением символьной константы является целое число, представляющее символ. Например, константа `'z'` представляет целочисленный код символа `z`, а константа `'\n'` – код символа перевода строки (122 и 10 в кодировке ASCII соответственно).

Строка – это последовательность символов, обрабатываемая как единое целое. Строка может включать буквы, цифры и различные специальные символы, такие как +, -, *, / или $. **Строковые литералы**, или **строковые константы**, в языке C записываются в двойных кавычках, как показано ниже:

```
"John Q. Doe"              (имя)
"99999 Main Street"        (адрес)
"Waltham, Massachusetts"   (город и штат)
"(201) 555-1212"           (номер телефона)
```

Строка в языке C является массивом символов, последний элемент которого содержит **нулевой символ** (`'\0'`). Доступ к строке осуществляется посредством *указателя* на первый символ. Значением строки является адрес первого ее символа. То есть можно сказать, что любая строка в языке C является указателем – указателем на ее первый символ. В этом смысле строки подобны массивам, потому что любой массив также является указателем на его первый элемент.

Массив символов, или *переменная типа* `char *`, может быть инициализирован строковым литералом. Следующие определения:

```
char color[] = "blue";
const char *colorPtr = "blue";
```

инициализируют переменные строкой `"blue"`. Первое определение создает массив `color` с пятью элементами, содержащими символы `'b'`, `'l'`, `'u'`, `'e'` и `'\0'`. Второе определение создает переменную-указатель `colorPtr`, которая ссылается на строку `"blue"`, находящуюся где-то в памяти.

Когда переменная типа `char *` инициализируется строковым литералом, некоторые компиляторы могут размещать саму строку в области памяти, недоступной для изменения. Если необходимо иметь возможность изменять строковые литералы во время выполнения, их следует сохранять в массивах символов, чтобы гарантировать доступность для изменения в любых системах.

Предыдущее определение массива можно записать иначе:

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

Когда определяется массив, предназначенный для хранения строки, он должен иметь достаточный размер, чтобы в нем разместились все символы строки плюс завершающий нулевой символ. Встретив предыдущее определение, компилятор автоматически определит размер массива, исходя из количества символов в списке инициализации.

Выделение пространства, недостаточного для хранения завершающего нулевого символа, является ошибкой.

Попытка вывести «строку», не завершающуюся нулевым символом, является ошибкой.

При сохранении строки в массиве символов убедитесь, что массив имеет достаточный размер, чтобы в нем можно было сохранить наибольшую возможную строку. Язык C не ограничивает размеры строк каким-либо фиксированным значением. Если строка окажется длиннее массива символов, куда она сохраняется, символы, вышедшие за пределы массива, могут затереть данные в области памяти, следующей за массивом.

Сохранить строку в массив можно с помощью функции `scanf`. Например, следующая инструкция сохранит строку в массив `word[20]`:

```
scanf( "%19s", word );
```

Строка, введенная пользователем, будет сохранена в `word`, где переменная `word` – это массив, имя которого, разумеется, является указателем, поэтому аргумент `word` не требуется предварять оператором `&`. В разделе 6.4 уже говорилось, что функция `scanf` будет читать символы, пока не встретит символа пробела, табуляции, перевода строки или признака конца файла. То есть если в спецификаторе преобразования `%19s` опустить ширину поля `19`, пользователь сможет ввести более 19 символов и вызвать аварийное завершение программы! По этой причине *всегда* следует явно указывать ширину поля в функции `scanf` при чтении строк в массивы символов. Ширина поля `19` в предыдущей инструкции гарантирует, что `scanf` прочитает *не более* 19 символов и в последнем элементе массива сохранит завершающий нулевой символ. Это не позволит записать символы за пределы массива. (Для чтения строк произвольной длины существует нестандартная, но широко поддерживаемая функция `readline`, прототип которой обычно находится в заголовочном файле `stdio.h`.) Чтобы массив символов можно было обрабатывать как строку, он обязательно должен содержать завершающий нулевой символ.

Попытка обработать символ как строку является ошибкой. Строка – это указатель, возможно большое целое число, тогда как символ – это небольшое целое число (коды ASCII занимают диапазон значений 0–255). Во многих системах такая попытка вызовет ошибку нарушения прав доступа, потому что младшие адреса памяти резервируются для специальных целей, например для размещения векторов перехода к обработчикам прерываний.

Попытка передать символ в качестве аргумента функции, ожидающей получить строку (и наоборот), вызовет ошибку во время компиляции.

8.3 Библиотека функций для работы с символами

Библиотека функций для работы с символами (`<ctype.h>`) включает несколько функций, которые могут пригодиться для проверки символьных данных и выполнения операций с ними. Каждая функция принимает аргумент типа `unsigned char` или значение `EOF`. Как уже рассказывалось в главе 4, символы часто интерпретируются как целые числа, потому что в языке C символы являются однобайтными целыми числами. `EOF` обычно имеет значение –1. В табл. 8.1 перечислены некоторые функции из библиотеки для работы с символами.

Таблица 8.1 | Функции из библиотеки для работы с символами (<ctype.h>)

Прототип	Описание
`int isblank( int c );`	Вернет истинное значение, если `c` – пробельный символ, способный служить разделителем слов в строке текста, и 0 (ложное значение) в противном случае. [*Обратите внимание*: эта функция недоступна в Microsoft Visual C++]
`int isdigit( int c );`	Вернет истинное значение, если `c` – цифра, и 0 (ложное значение) в противном случае
`int isalpha( int c );`	Вернет истинное значение, если `c` – буква, и 0 (ложное значение) в противном случае
`int isalnum( int c );`	Вернет истинное значение, если `c` – буква или цифра, и 0 (ложное значение) в противном случае
`int isxdigit( int c );`	Вернет истинное значение, если `c` – шестнадцатеричная цифра, и 0 (ложное значение) в противном случае. (Подробное описание двоичных, восьмеричных, десятичных и шестнадцатеричных чисел приводится в приложении C)
`int islower( int c );`	Вернет истинное значение, если `c` – буква нижнего регистра, и 0 (ложное значение) в противном случае

Таблица 8.1 | (*окончание*)

Прототип	Описание
`int isupper( int c );`	Вернет истинное значение, если `c` – буква верхнего регистра, и 0 (ложное значение) в противном случае
`int tolower( int c );`	Если `c` – буква верхнего регистра, функция `tolower` вернет букву нижнего регистра. В противном случае вернет исходный аргумент
`int toupper( int c );`	Если `c` – буква нижнего регистра, функция `tolower` вернет букву верхнего регистра. В противном случае вернет исходный аргумент
`int isspace( int c );`	Вернет истинное значение, если `c` является пробельным символом (перевод строки (`'\n'`), пробел (`' '`), перевод формата (`'\f'`), возврат каретки (`'\r'`), горизонтальная табуляция (`'\t'`) или вертикальная табуляция (`'\v'`)), и 0 (ложное значение) в противном случае
`int iscntrl( int c );`	Вернет истинное значение, если `c` – управляющий символ, и 0 (ложное значение) в противном случае
`int ispunct( int c );`	Вернет истинное значение, если `c` – печатаемый символ, не являющийся пробельным символом, цифрой или буквой, и 0 (ложное значение) в противном случае
`int isprint( int c );`	Вернет истинное значение, если `c` – печатаемый символ, включая пробел, и 0 (ложное значение) в противном случае
`int isgraph( int c );`	Вернет истинное значение, если `c` – печатаемый символ, исключая пробел, и 0 (ложное значение) в противном случае

8.3.1 Функции isdigit, isalpha, isalnum и isxdigit

В примере 8.1 демонстрируется использование функций `isdigit`, `isalpha`, `isalnum` и `isxdigit`. Функция `isdigit` определяет, является ли ее аргумент цифрой (0–9). Функция `isalpha` определяет, является ли ее аргумент буквой верхнего (A–Z) или нижнего (a–z) регистра. Функция `isalnum` определяет, является ли ее аргумент буквой или цифрой. Функция `isxdigit` определяет, является ли ее аргумент **шестнадцатеричной цифрой** (A–F, a–f, 0–9).

Пример 8.1 | Демонстрация функций isdigit, isalpha, isalnum и isxdigit

```
1 // Пример 8.1: fig08_02.c
2 // Демонстрация функций isdigit, isalpha, isalnum и isxdigit
3 #include <stdio.h>
4 #include <ctype.h>
5
6 int main( void )
7 {
8    printf( "%s\n%s%s\n%s%s\n\n", "According to isdigit: ",
9       isdigit( '8' ) ? "8 is a " : "8 is not a ", "digit",
```

```
10       isdigit( '#' ) ? "# is a " : "# is not a ", "digit" );
11
12    printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
13       "According to isalpha:",
14       isalpha( 'A' ) ? "A is a " : "A is not a ", "letter",
15       isalpha( 'b' ) ? "b is a " : "b is not a ", "letter",
16       isalpha( '&' ) ? "& is a " : "& is not a ", "letter",
17       isalpha( '4' ) ? "4 is a " : "4 is not a ", "letter" );
18
19    printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
20       "According to isalnum:",
21       isalnum( 'A' ) ? "A is a " : "A is not a ",
22       "digit or a letter",
23       isalnum( '8' ) ? "8 is a " : "8 is not a ",
24       "digit or a letter",
25       isalnum( '#' ) ? "# is a " : "# is not a ",
26       "digit or a letter" );
27
28    printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
29       "According to isxdigit:",
30       isxdigit( 'F' ) ? "F is a " : "F is not a ",
31       "hexadecimal digit",
32       isxdigit( 'J' ) ? "J is a " : "J is not a ",
33       "hexadecimal digit",
34       isxdigit( '7' ) ? "7 is a " : "7 is not a ",
35       "hexadecimal digit",
36       isxdigit( '$' ) ? "$ is a " : "$ is not a ",
37       "hexadecimal digit",
38       isxdigit( 'f' ) ? "f is a " : "f is not a ",
39       "hexadecimal digit" );
40 } // конец main
```

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Для выбора между строками `" is a "` (является) и `" is not a "` (не является) при выводе результатов проверки в примере 8.1 используется условный оператор (`?:`). Например, выражение

```
isdigit( '8' ) ? "8 is a " : "8 is not a "
```

выведет строку "8 is a ", если '8' является цифрой, и "8 is not a ", если '8' не является цифрой (то есть если isdigit вернет 0).

8.3.2 Функции islower, isupper, tolower и toupper

В примере 8.2 демонстрируется использование функций islower, isupper, tolower и toupper. Функция islower определяет, является ли ее аргумент буквой нижнего регистра (a–z). Функция isupper определяет, является ли ее аргумент буквой верхнего регистра (A–Z). Функция tolower преобразует букву верхнего регистра в букву нижнего регистра и возвращает результат. Если аргумент не является буквой верхнего регистра, tolower вернет исходный аргумент. Функция toupper преобразует букву нижнего регистра в букву верхнего регистра и возвращает результат. Если аргумент не является буквой нижнего регистра, toupper вернет исходный аргумент.

Пример 8.2 | Демонстрация функций islower, isupper, tolower, toupper

```
// Пример 8.2: fig08_03.c
// Демонстрация функций islower, isupper, tolower, toupper
#include <stdio.h>
#include <ctype.h>

int main( void )
{
   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to islower:",
      islower( 'p' ) ? "p is a " : "p is not a ",
      "lowercase letter",
      islower( 'P' ) ? "P is a " : "P is not a ",
      "lowercase letter",
      islower( '5' ) ? "5 is a " : "5 is not a ",
      "lowercase letter",
      islower( '!' ) ? "! is a " : "! is not a ",
      "lowercase letter" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to isupper:",
      isupper( 'D' ) ? "D is an " : "D is not an ",
      "uppercase letter",
      isupper( 'd' ) ? "d is an " : "d is not an ",
      "uppercase letter",
      isupper( '8' ) ? "8 is an " : "8 is not an ",
      "uppercase letter",
      isupper( '$' ) ? "$ is an " : "$ is not an ",
      "uppercase letter" );

   printf( "%s%c\n%s%c\n%s%c\n%s%c\n",
      "u converted to uppercase is ", toupper( 'u' ),
      "7 converted to uppercase is ", toupper( '7' ),
      "$ converted to uppercase is ", toupper( '$' ),
```

```
34       "L converted to lowercase is ", tolower( 'L' ));
35 } // конец main
```

```
According to islower:
p is a lowercase letter
P is not a lowercase letter
5 is not a lowercase letter
! is not a lowercase letter

According to isupper:
D is an uppercase letter
d is not an uppercase letter
8 is not an uppercase letter
$ is not an uppercase letter

u converted to uppercase is U
7 converted to uppercase is 7
$ converted to uppercase is $
L converted to lowercase is l
```

8.3.3 Функции isspace, iscntrl, ispunct, isprint и isgraph

В примере 8.3 демонстрируется использование функций `isspace`, `iscntrl`, `ispunct`, `isprint` и `isgraph`.

Функция `isspace` определяет, является ли ее аргумент одним из следующих пробельных символов: пробел (`' '`), перевод формата (`'\f'`), перевод строки (`'\n'`), возврат каретки (`'\r'`), горизонтальная табуляция (`'\t'`) и вертикальная табуляция (`'\v'`). Функция `iscntrl` определяет, является ли ее аргумент одним из следующих **управляющих символов**: горизонтальная табуляция (`'\t'`), вертикальная табуляция (`'\v'`), перевод формата (`'\f'`), сигнал (`'\a'`), забой (`'\b'`), возврат каретки (`'\r'`) и перевод строки (`'\n'`). Функция `ispunct` определяет, является ли ее аргумент **печатаемым символом**, отличным от пробела, цифры и буквы, таким как `$`, `#`, `(`, `)`, `[`, `]`, `{`, `}`, `;`, `:` и `%`. Функция `isprint` определяет, является ли ее аргумент отображаемым на экране символом (включая пробел). Функция `isgraph` подобна функции `isprint`, но исключает символ пробела.

Пример 8.3 | Демонстрация функций isspace, iscntrl, ispunct, isprint и isgraph

```
 1 // Пример 8.3: fig08_04.c
 2 // Демонстрация функций isspace, iscntrl, ispunct, isprint и isgraph
 3 #include <stdio.h>
 4 #include <ctype.h>
 5
 6 int main( void )
 7 {
 8    printf( "%s\n%s%s%s\n%s%s%s\n%s%s\n\n",
 9       "According to isspace:",
10       "Newline", isspace( '\n' ) ? " is a " : " is not a ",
```

```
11       "whitespace character", "Horizontal tab",
12       isspace( '\t' ) ? " is a " : " is not a ",
13       "whitespace character",
14       isspace( '%' ) ? "% is a " : "% is not a ",
15       "whitespace character" );
16
17    printf( "%s\n%s%s%s\n%s%s\n\n", "According to iscntrl:",
18       "Newline", iscntrl( '\n' ) ? " is a " : " is not a ",
19       "control character", iscntrl( '$' ) ? "$ is a " :
20       "$ is not a ", "control character" );
21
22    printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
23       "According to ispunct:",
24       ispunct( ';' ) ? "; is a " : "; is not a ",
25       "punctuation character",
26       ispunct( 'Y' ) ? "Y is a " : "Y is not a ",
27       "punctuation character",
28       ispunct( '#' ) ? "# is a " : "# is not a ",
29       "punctuation character" );
30
31    printf( "%s\n%s%s\n%s%s%s\n\n", "According to isprint:",
32       isprint( '$' ) ? "$ is a " : "$ is not a ",
33       "printing character",
34       "Alert", isprint( '\a' ) ? " is a " : " is not a ",
35       "printing character" );
36
37    printf( "%s\n%s%s\n%s%s%s\n", "According to isgraph:",
38       isgraph( 'Q' ) ? "Q is a " : "Q is not a ",
39       "printing character other than a space",
40       "Space", isgraph( ' ' ) ? " is a " : " is not a ",
41       "printing character other than a space" );
42 } // конец main
```

```
According to isspace:
Newline is a whitespace character
Horizontal tab is a whitespace character
% is not a whitespace character

According to iscntrl:
Newline is a control character
$ is not a control character

According to ispunct:
; is a punctuation character
Y is not a punctuation character
# is a punctuation character

According to isprint:
$ is a printing character
Alert is not a printing character

According to isgraph:
Q is a printing character other than a space
Space is not a printing character other than a space
```

8.4 Функции преобразования строк

В этом разделе будут представлены функции преобразования строк из библиотеки `<stdlib.h>`. С помощью этих функций можно преобразовывать строки цифр в целые и вещественные числа. В табл. 8.2 перечислены некоторые функции преобразования строк. Стандарт языка C дополнительно определяет функции `strtoll` и `strtoull`, преобразующие строки в значения типа `long long int` и `unsigned long long int` соответственно. Обратите внимание на использование квалификатора `const` в объявлении параметра `nPtr` в заголовках функций (читается справа налево как «`nPtr` – это указатель на символьную константу»); квалификатор `const` указывает, что значение аргумента не изменяется.

Таблица 8.2 | Функции преобразования строк

Прототип	Описание
`double strtod( const char *nPtr, char **endPtr );`	Преобразует строку `nPtr` в значение типа `double`
`long strtol( const char *nPtr, char **endPtr, int base );`	Преобразует строку `nPtr` в значение типа `long`
`unsigned long strtoul( const char *nPtr, char **endPtr, int base );`	Преобразует строку `nPtr` в значение типа `unsigned long`

8.4.1 Функция strtod

Функция `strtod` (пример 8.4) преобразует последовательность символов, представляющую вещественное число, в значение типа `double`. Она возвращает 0, если никакая часть первого аргумента не может быть преобразована в число. Функция принимает два аргумента – строку (`char *`) и указатель на строку (`char **`). В первом аргументе ей передается последовательность символов для преобразования – любые пробельные символы в начале и в конце строки игнорируются. Во втором аргументе `char **` передается указатель на переменную типа `char *` в вызывающей функции (`stringPtr`), куда помещается указатель на символ в исходной строке, следующий сразу же за преобразованным фрагментом, или на всю строку, если никакая ее часть не может быть преобразована. Строка 14

```
d = strtod( string, &stringPtr );
```

присвоит переменной `d` значение типа `double`, полученное преобразованием фрагмента исходной строки `string`, а переменной `stringPtr` будет присвоен адрес первого символа в исходной строке `string`, следующего сразу за преобразованным фрагментом (`51.2`).

Пример 8.4 | Демонстрация функции strtod

```
 1 // Пример 8.4: fig08_06.c
 2 // Демонстрация функции strtod
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 int main( void )
 7 {
 8    // инициализировать указатель string
 9    const char *string = "51.2% are admitted"; // инициализация исходной
                                                 // строки
10
11    double d;        // переменная для хранения результата преобразования
12    char *stringPtr; // создать указатель на символ
13
14    d = strtod( string, &stringPtr );
15
16    printf( "The string \"%s\" is converted to the\n", string );
17    printf( "double value %.2f and the string \"%s\"\n", d, stringPtr );
18 } // конец main
```

```
The string "51.2% are admitted" is converted to the
double value 51.20 and the string "% are admitted"
```

8.4.2 Функция strtol

Функция `strtol` (пример 8.5) преобразует последовательность символов, представляющую целое число, в значение типа `long int`. Она возвращает 0, если никакая часть первого аргумента не может быть преобразована в число. Функция принимает три аргумента – строку (`char *`), указатель на строку и целое число. В первом аргументе ей передается последовательность символов для преобразования – любые пробельные символы в начале и в конце строки игнорируются. Во втором аргументе `char **` передается указатель на переменную типа `char *` в вызывающей функции (`remainderPtr`), куда помещается указатель на символ в исходной строке, следующий сразу же за преобразованным фрагментом, или на всю строку, если никакая ее часть не может быть преобразована. Третий аргумент определяет основание системы счисления. Строка 13

```
x = strtol( string, &remainderPtr, 0 );
```

присвоит переменной `x` значение типа `long int`, полученное преобразованием фрагмента исходной строки `string`, а переменной `remainderPtr` будет присвоен адрес первого символа в исходной строке `string`, следующего сразу за преобразованным фрагментом. Если во втором аргументе передать `NULL`, он просто будет проигнорирован. Значение 0 в третьем аргументе указывает, что исходная строка может содержать строковое представление числа в восьмеричной системе счисления (с основанием 8), десятичной (с основанием 10) или шестнадцатеричной (с основанием 16). Основание

системы счисления может быть любым целым числом в диапазоне от 2 до 36 или 0. (Подробнее о восьмеричной, десятичной и шестнадцатеричной системах счисления рассказывается в приложении C.) Для представления целых чисел в системах счисления с основанием от 11 до 36, кроме цифр, используются также символы A–Z, представляющие значения от 10 до 35. Например, строковое представление шестнадцатеричного числа может содержать цифры 0–9 и символы A–F. Строковое представление числа в системе счисления с основанием 11 может содержать цифры 0–9 и символ A. Строковое представление числа в системе счисления с основанием 24 может содержать цифры 0–9 и символы A–N. Строковое представление числа в системе счисления с основанием 36 может содержать цифры 0–9 и символы A–Z. Функция возвращает 0, если никакая часть исходной строки не может быть преобразована в значение типа `long int`.

Пример 8.5 | Демонстрация функции strtol

```
// Пример 8.5: fig08_07.c
// Демонстрация функции strtol
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
   const char *string = "-1234567abc"; // инициализировать указатель
                                       // string

   char *remainderPtr; // создать указатель на символ
   long x;             // переменная для хранения результата преобразования

   x = strtol( string, &remainderPtr, 0 );

   printf( "%s\"%s\"\n%s%ld\n%s\"%s\"\n%s%ld\n",
      "The original string is ", string,
      "The converted value is ", x,
      "The remainder of the original string is ",
      remainderPtr,
      "The converted value plus 567 is ", x + 567 );
} // конец main
```

```
The original string is "-1234567abc"
The converted value is -1234567
The remainder of the original string is "abc"
The converted value plus 567 is -1234000
```

8.4.3 Функция strtoul

Функция `strtoul` (пример 8.6) преобразует последовательность символов, представляющую целое число, в значение типа `unsigned long int`. Эта функция совершенно идентична функции `strtol`. Инструкция

```
x = strtoul( string, &remainderPtr, 0 );
```

в строке 12 примера 8.6 присвоит переменной `x` значение типа `unsigned long int`, полученное преобразованием фрагмента исходной строки `string`, а переменной `remainderPtr` будет присвоен адрес первого символа в исходной строке `string`, следующего сразу за преобразованным фрагментом. Значение 0 в третьем аргументе указывает, что исходная строка может содержать строковое представление числа в восьмеричной, десятичной или шестнадцатеричной системе счисления.

Пример 8.6 | Демонстрация функции strtoul

```
 1 // Пример 8.6: fig08_08.c
 2 // Демонстрация функции strtoul
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 int main( void )
 7 {
 8    const char *string = "1234567abc"; // инициализировать указатель
                                         // string
 9    unsigned long int x; // переменная для хранения результата
                           // преобразования
10    char *remainderPtr;  // создать указатель на символ
11
12    x = strtoul( string, &remainderPtr, 0 );
13
14    printf( "%s\"%s\"\n%s%lu\n%s\"%s\"\n%s%lu\n",
15       "The original string is ", string,
16       "The converted value is ", x,
17       "The remainder of the original string is ",
18       remainderPtr,
19       "The converted value minus 567 is ", x - 567 );
20 } // конец main
```

```
The original string is "1234567abc"
The converted value is 1234567
The remainder of the original string is "abc"
The converted value minus 567 is 1234000
```

8.5 Стандартная библиотека ввода/вывода

В этом разделе будут представлены функции из стандартной библиотеки ввода/вывода (`<stdio.h>`), предназначенные для работы с символьными и строковыми данными. В табл. 8.3 перечислены некоторые функции из стандартной библиотеки ввода/вывода.

8.5.1 Функции fgets и putchar

В примере 8.7 демонстрируется использование функций `fgets` и `putchar` для чтения строки текста из стандартного ввода (с клавиатуры) и вывода последовательности символов в обратном направлении. Функция `fgets` читает символы из *стандартного ввода* в свой первый аргумент – массив символов,

Таблица 8.3 | Функции для работы с символами и строками из стандартной библиотеки ввода/вывода

Прототип	Описание
`int getchar( void );`	Вводит следующий символ из стандартного ввода и возвращает его в виде целого числа
`char *fgets( char *s, int n, FILE *stream );`	Вводит символы из указанного потока `stream` в массив `s`, пока не встретит символ перевода строки или признак конца файла или пока не будут прочитаны `n - 1` байт. В этой главе мы будем использовать поток `stdin` – стандартный поток ввода, который обычно применяется для чтения символов, вводимых с клавиатуры. После ввода последнего символа эта функция добавляет в конец массива нулевой символ. Возвращает строку в `s`
`int putchar( int c );`	Выводит символ, представленный аргументом `c`, и возвращает его же в виде целого числа
`int puts( const char *s );`	Выводит строку `s` и завершает вывод символом перевода строки. Возвращает ненулевое целочисленное значение в случае успеха и `EOF` в случае ошибки
`int sprintf( char *s, const char *format, ... );`	Действует подобно функции `printf`, но результат не выводится на экран, а записывается в массив `s`. Возвращает количество символов, сохраненных в `s`, или `EOF` в случае ошибке. [*Обратите внимание*: в приложении E и в разделе «Безопасное программирование на C» мы расскажем о родственных ей и более безопасных функциях `snprintf` и `snprintf_s`]
`int sscanf( char *s, const char *format, ... );`	Действует подобно функции `scanf`, за исключением того, что исходные данные читаются не с клавиатуры, а из массива `s`. Возвращает количество удачно прочтенных элементов или `EOF` в случае ошибки

пока не встретит символ *перевода строки* либо признак *конца файла* или пока не будет прочитано максимально возможное количество символов. Максимальное количество символов на единицу меньше числа, передаваемого функции `fgets` во втором аргументе. Третий аргумент определяет поток для чтения – в данном примере используется поток стандартного ввода (`stdin`). По завершении чтения в массив автоматически добавляется нулевой символ (`'\0'`). Функция `putchar` выводит символ, переданный ей в аргументе. Программа *рекурсивно* вызывает функцию `reverse` для вывода строки текста в обратном порядке. Если первый символ в массиве, переданном функции

reverse, является нулевым символом '\0', reverse просто возвращает управление. В противном случае она вызывает себя, передавая адрес фрагмента массива, начинающегося с элемента sPtr[1], а когда происходит возврат из рекурсивного вызова, выводит символ из sPtr[0] с помощью putchar. Такой порядок следования инструкций в ветке else условной инструкции if обеспечивает обход строки в обратном направлении, прежде чем текущий символ будет выведен.

Пример 8.7 | Демонстрация функций fgets и putchar

```
// Пример 8.7: fig08_10.c
// Демонстрация функций fgets и putchar
#include <stdio.h>
#define SIZE 80

void reverse( const char * const sPtr ); // прототип

int main( void )
{
   char sentence[ SIZE ]; // создать массив символов

   puts( "Enter a line of text:" );

   // прочитать строку текста с помощью fgets
   fgets( sentence, SIZE, stdin );

   puts( "\nThe line printed backward is:" );
   reverse( sentence );
} // конец main

// рекурсивно выводит символы в строке в обратном порядке
void reverse( const char * const sPtr )
{
   // если встречен конец строки
   if ( '\0' == sPtr[ 0 ] ) { // базовый случай
      return;
   } // конец if
   else { // если не конец строки
      reverse( &sPtr[ 1 ] );  // шаг рекурсии
      putchar( sPtr[ 0 ] );   // вывести с помощью putchar
   } // конец else
} // конец функции reverse
```

```
Enter a line of text:
Characters and Strings

The line printed backward is:
sgnirtS dna sretcarahC

Enter a line of text:
able was I ere I saw elba

The line printed backward is:
able was I ere I saw elba
```

8.5.2 Функция getchar

В примере 8.8 демонстрируется использование функций `getchar` и `puts` для чтения символов из стандартного ввода в массив символов `sentence` и отображения их в виде строки. Функция `getchar` читает символ из *стандартного ввода* и возвращает его в виде целого числа. Как вы уже знаете, функция `puts` принимает строку в качестве аргумента и выводит ее, добавляя в конец символ *перевода строки*. Программа прекращает ввод символов, либо когда их количество становится равным 79, либо когда `getchar` прочитает символ перевода строки, введенный пользователем. В массив `sentence` добавляется нулевой символ (строка 20), поэтому его можно использовать как строку. В строке 24 вызывается функция `puts` для вывода строки, хранящейся в массиве `sentence`.

Пример 8.8 | Демонстрация функции getchar

```
1 // Пример 8.8: fig08_11.c
2 // Демонстрация функции getchar.
3 #include <stdio.h>
4 #define SIZE 80
5
6 int main( void )
7 {
8    int c; // переменная для хранения символа, введенного пользователем
9    char sentence[ SIZE ]; // создать массив символов
10   int i = 0;             // инициализировать счетчик i
11
12   // предложить пользователю ввести строку текста
13   puts( "Enter a line of text:" );
14
15   // прочитать каждый символ отдельным вызовом getchar
16   while ( i < SIZE - 1 && ( c = getchar() ) != '\n' ) {
17      sentence[ i++ ] = c;
18   } // конец while
19
20   sentence[ i ] = '\0'; // завершить строку
21
22   // вывести содержимое sentence с помощью puts
23   puts( "\nThe line entered was:" );
24   puts( sentence );
25 } // конец main
```

```
Enter a line of text:
This is a test.

The line entered was:
This is a test.
```

8.5.3 Функция sprintf

В примере 8.9 демонстрируется использование функции `sprintf` для вывода форматированных данных в массив `s` – массив символов. Функция позволяет использовать те же спецификаторы преобразований, что и `printf`

(подробнее о форматировании данных рассказывается в главе 9). Программа вводит два числа, целое и вещественное, и выводит их в массив s. Массив s передается функции sprintf в первом аргументе. [*Обратите внимание*: если ваша система поддерживает snprintf_s, вместо sprintf предпочтительнее использовать эту функцию. Если ваша система не поддерживает функцию snprintf_s, но поддерживает snprintf, тогда лучше использовать ее. Обе эти функции мы подробнее рассмотрим в приложении E.]

Пример 8.9 | Демонстрация функции sprintf

```
// Пример 8.9: fig08_12.c
// Демонстрация функции sprintf
#include <stdio.h>
#define SIZE 80

int main( void )
{
   char s[ SIZE ]; // создать массив символов
   int x;          // для ввода целого числа
   double y;       // для ввода вещественного числа

   puts( "Enter an integer and a double:" );
   scanf( "%d%lf", &x, &y );

   sprintf( s, "integer:%6d\ndouble:%8.2f", x, y );

   printf( "%s\n%s\n",
           "The formatted output stored in array s is:", s );
} // конец main
```

```
Enter an integer and a double:
298 87.375
The formatted output stored in array s is:
integer:   298
double:   87.38
```

8.5.4 Функция sscanf

В примере 8.10 демонстрируется использование функции sscanf для чтения форматированных данных из массива символов s. Функция позволяет использовать те же спецификаторы преобразований, что и scanf. Программа вводит два числа, целое и вещественное, из массива s и сохраняет их значения в переменных x и y соответственно. Затем она выводит значения переменных x и y. Массив s передается функции sscanf в первом аргументе.

Пример 8.10 | Демонстрация функции sscanf

```
// Пример 8.10: fig08_13.c
// Демонстрация функции sscanf
#include <stdio.h>

int main( void )
{
```

```
 7    char s[] = "31298 87.375"; // инициализировать массив s
 8    int x;                     // для ввода целого числа
 9    double y;                  // для ввода вещественного числа
10
11    sscanf( s, "%d%lf", &x, &y );
12    printf( "%s\n%s%6d\n%s%8.3f\n",
13       "The values stored in character array s are:",
14       "integer:", x, "double:", y );
15 } // конец main
```

```
The values stored in character array s are:
integer: 31298
double: 87.375
```

8.6 Функции для работы со строками

Библиотека функций для работы со строками (`<string.h>`) предоставляет множество удобных функций, позволяющих выполнять операции со строковыми данными (копировать и объединять строки), сравнивать строки, производить поиск символов и других строк в строках, разбивать строки на лексемы (логические фрагменты) и определять длину строк. В этом разделе будут представлены некоторые функции из библиотеки для работы со строками. В табл. 8.4 перечислены также функции из этой библиотеки. Каждая функция – кроме `strncpy` – добавляет нулевой символ в конец результата. [*Обратите внимание*: для всех этих функций имеются более безопасные версии, определяемые приложением Annex K к стандарту C11. Мы представим их в разделе «Безопасное программирование на C» в конце этой главы, а также в приложении E.]

Таблица 8.4 | Функции для работы со строками

Прототип	Описание
`char *strcpy( char *s1, const char *s2 )`	Копирует строку `s2` в массив `s1`. Возвращает `s1`
`char *strncpy( char *s1, const char *s2, size_t n )`	Копирует не более `n` символов из строки `s2` в массив `s1`. Возвращает `s1`
`char *strcat( char *s1, const char *s2 )`	Добавляет строку `s2` в конец массива `s1`. Первый символ строки `s2` затирает завершающий нулевой символ в массиве `s1`. Возвращает `s1`
`char *strncat( char *s1, const char *s2, size_t n )`	Добавляет не более `n` символов из строки `s2` в конец массива `s1`. Первый символ строки `s2` затирает завершающий нулевой символ в массиве `s1`. Возвращает `s1`

Функции `strncpy` и `strncat` имеют дополнительный параметр типа `size_t`. Функция `strcpy` копирует свой второй аргумент (строку) в свой

первый аргумент – массив символов, который *должен иметь достаточный размер*, чтобы вместить строку и завершающий нулевой символ, который также копируется из исходной строки. Функция `strncpy` действует подобно функции `strcpy`, за исключением того, что `strncpy` позволяет указать максимальное количество символов, которые могут быть скопированы из строки в массив. *Функция `strncpy` не всегда копирует завершающий нулевой символ из своего второго аргумента. Это происходит, только если указанное максимальное количество копируемых символов больше, хотя бы на единицу длины строки.* Например, если во втором аргументе передать строку `"test"`, завершающий нулевой символ будет скопирован, только если в третьем аргументе функции `strncpy` передать значение не ниже `5` (четыре символа в строке `"test"` плюс завершающий нулевой символ). Если третий аргумент будет иметь значение больше `5`, нулевые символы будут продолжать записываться в массив, пока общее количество скопированных символов не достигнет значения, указанного в третьем аргументе.

При использовании функций для работы со строками подключайте заголовочный файл `<string.h>`.

Если забыть добавить завершающий нулевой символ в конец первого аргумента функции `strncpy`, когда значение третьего аргумента меньше или равно длине строки во втором аргументе, это может привести к ошибке во время выполнения.

8.6.1 Функции strcpy и strncpy

В примере 8.11 демонстрируется использование функции `strcpy` для копирования целой строки `x` в массив `y` и использование функции `strncpy` для копирования первых 14 символов из массива `x` в массив `z`. Программа в примере предусматривает добавление *нулевого символа* (`'\0'`) в конец массива `z`, потому что вызов `strncpy` *не приводит к копированию нулевого символа* (значение третьего аргумента меньше длины строки во втором аргументе).

Пример 8.11 | Демонстрация функций strcpy и strncpy

```
 1 // Пример 8.11: fig08_15.c
 2 // Демонстрация функций strcpy и strncpy
 3 #include <stdio.h>
 4 #include <string.h>
 5 #define SIZE1 25
 6 #define SIZE2 15
 7
 8 int main( void )
 9 {
10    char x[] = "Happy Birthday to You"; // инициализировать массив
                                          // символов x
```

```
   char y[ SIZE1 ]; // создать массив y
   char z[ SIZE2 ]; // создать массив z

   // скопировать содержимое x в y
   printf( "%s%s\n%s%s\n",
      "The string in array x is: ", x,
      "The string in array y is: ", strcpy( y, x ) );

   // скопировать первые 14 символов из x в z.
   // Нулевой символ не копируется.
   strncpy( z, x, SIZE2 - 1 );

   z[ SIZE2 - 1 ] = '\0'; // добавить завершающий нулевой символ в z
   printf( "The string in array z is: %s\n", z );
} // конец main
```

```
The string in array x is: Happy Birthday to You
The string in array y is: Happy Birthday to You
The string in array z is: Happy Birthday
```

8.6.2 Функции strcat и strncat

Функция `strcat` добавляет свой второй аргумент (строку) в конец первого аргумента (массив символов, содержащий строку). Первый символ строки во втором аргументе затирает завершающий нулевой символ (`'\0'`) в первом аргументе. *Массив, используемый для хранения первой строки, должен иметь достаточный размер, чтобы вместить вторую строку и завершающий нулевой символ.* Функция `strncat` добавляет указанное в третьем аргументе количество символов из строки во втором аргументе в конец строки в первом аргументе. Завершающий нулевой символ автоматически добавляется в результат. Пример 8.12 демонстрирует использование функций `strcat` и `strncat`.

Пример 8.12 | Демонстрация функций strcat и strncat

```
// Пример 8.12: fig08_16.c
// Демонстрация функций strcat и strncat
#include <stdio.h>
#include <string.h>

int main( void )
{
   char s1[ 20 ] = "Happy "; // инициализировать массив символов s1
   char s2[] = "New Year ";  // инициализировать массив символов s2
   char s3[ 40 ] = "";       // инициализировать пустой массив символов s3

   printf( "s1 = %s\ns2 = %s\n", s1, s2 );

   // добавить s2 в конец s1
   printf( "strcat( s1, s2 ) = %s\n", strcat( s1, s2 ) );

   // добавить первые 6 символов из s1 в s3.
   // Добавить '\0' после последнего символа.
```

```
19    printf( "strncat( s3, s1, 6 ) = %s\n", strncat( s3, s1, 6 ) );
20
21    // добавить s1 в конец s3
22    printf( "strcat( s3, s1 ) = %s\n", strcat( s3, s1 ) );
23 } // конец main
```

```
s1 = Happy
s2 = New Year
strcat( s1, s2 ) = Happy New Year
strncat( s3, s1, 6 ) = Happy
strcat( s3, s1 ) = Happy Happy New Year
```

8.7 Функции сравнения строк

В этом разделе будут представлены функции сравнения строк, `strcmp` и `strncmp`. В табл. 8.5 указаны прототипы и дается краткое описание каждой из них.

Таблица 8.5 | Функции для работы со строками

Прототип	Описание
`int strcmp( const char *s1, const char *s2 );`	Сравнивает строку `s1` со строкой `s2`. Функция возвращает 0, значение меньше или больше 0, если строка `s1` равна, меньше или больше строки `s2` соответственно
`int strncmp( const char *s1, const char *s2, size_t n );`	Сравнивает не более `n` первых символов из строки `s1` со строкой `s2`. Функция возвращает 0, значение меньше или больше 0, если строка `s1` равна, меньше или больше строки `s2` соответственно

Программа в примере 8.13 сравнивает три строки с помощью `strcmp` и `strncmp`. Функция `strcmp` сравнивает символ за символом строку в первом аргументе со строкой во втором аргументе. Она возвращает 0, если строки равны, *отрицательное значение* – если первая строка меньше второй, и *положительное значение* – если больше. Функция `strncmp` действует подобно функции `strcmp`, но сравнивает не более указанного количества символов. Функция `strncmp` *не* сравнивает символы, находящиеся за завершающим нулевым символом. Программа выводит числа, полученные в результате вызова каждой функции.

Пример 8.13 | Демонстрация функций strcmp and strncmp

```
1 // Пример 8.13: fig08_18.c
2 // Демонстрация функций strcmp and strncmp
3 #include <stdio.h>
4 #include <string.h>
5
```

```
 6 int main( void )
 7 {
 8    const char *s1 = "Happy New Year"; // инициализировать указатель
 9    const char *s2 = "Happy New Year"; // инициализировать указатель
10    const char *s3 = "Happy Holidays"; // инициализировать указатель
11
12    printf("%s%s\n%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n\n",
13       "s1 = ", s1, "s2 = ", s2, "s3 = ", s3,
14       "strcmp(s1, s2) = ", strcmp( s1, s2 ),
15       "strcmp(s1, s3) = ", strcmp( s1, s3 ),
16       "strcmp(s3, s1) = ", strcmp( s3, s1 ) );
17
18    printf("%s%2d\n%s%2d\n%s%2d\n",
19       "strncmp(s1, s3, 6) = ", strncmp( s1, s3, 6 ),
20       "strncmp(s1, s3, 7) = ", strncmp( s1, s3, 7 ),
21       "strncmp(s3, s1, 7) = ", strncmp( s3, s1, 7 ) );
22 } // конец main
```

```
s1 = Happy New Year
s2 = Happy New Year
s3 = Happy Holidays

strcmp(s1, s2) =  0
strcmp(s1, s3) =  1
strcmp(s3, s1) = -1

strncmp(s1, s3, 6) =  0
strncmp(s1, s3, 7) =  6
strncmp(s3, s1, 7) = -6
```

Предположение, что функции `strcmp` *и* `strncmp` *возвращают 1, когда аргументы равны, является логической ошибкой. Обе функции возвращают 0 (что довольно странно, если учесть, что 0 в языке C интерпретируется как ложное значение) в случае равенства строк. Поэтому при проверке равенства двух строк результат функции* `strcmp` *или* `strncmp` *следует сравнивать с 0.*

Только чтобы понять, что означают для строк понятия «больше», «меньше» и «равно», представьте процедуру сортировки списка фамилий в алфавитном порядке. Уверен, вы не колеблясь поместили бы фамилию «Jones» перед фамилией «Smith», потому что в алфавите первая буква в фамилии «Jones» предшествует первой букве в фамилии «Smith». Но алфавит – это не только список из 26 букв[1], это упорядоченный список символов. Каждая буква занимает в этом списке определенное положение. Буква «Z» – это не просто буква в алфавите; «Z» – это 26-я буква алфавита.

Откуда функции сравнения строк берут информацию о порядке следования букв в алфавите? Дело в том, что все символы в компьютере представлены числовыми кодами в определенной кодировке, такой как ASCII или Unicode; когда компьютер сравнивает две строки, он в действительности сравнивает числовые коды символов, составляющих строки.

[1] Здесь имеется в виду латинский алфавит. – *Прим. перев.*

8.8 Функции поиска в строках

В этом разделе будут представлены функции поиска в строках символов и других строках. Некоторые из имеющихся функций перечислены в табл. 8.6. Функции `strcspn` и `strspn` возвращают значение типа `size_t`. [*Обратите внимание*: для функции `strtok` имеется более безопасная версия, определяемая приложением Annex K к стандарту C11. Мы представим ее в разделе «Безопасное программирование на C» в конце этой главы, а также в приложении E.]

Таблица 8.6 | Функции поиска в строках

Прототип	Описание
`char *strchr( const char *s, int c );`	Ищет первое вхождение символа `c` в строке `s`. В случае успеха возвращает указатель на символ `c`. В противном случае возвращает `NULL`
`size_t strcspn( const char *s1, const char *s2 );`	Определяет и возвращает длину начального фрагмента строки `s1`, не содержащего символы из строки `s2`
`size_t strspn( const char *s1, const char *s2 );`	Определяет и возвращает длину начального фрагмента строки `s1`, содержащего только символы из строки `s2`
`char *strpbrk( const char *s1, const char *s2 );`	Ищет в строке `s1` первое вхождение любого символа из строки `s2`. В случае успеха возвращает указатель на символ в строке `s1`. В противном случае возвращает `NULL`
`char *strrchr( const char *s, int c );`	Ищет последнее вхождение символа `c` в строке `s`. В случае успеха возвращает указатель на символ `c`. В противном случае возвращает `NULL`
`char *strstr( const char *s1, const char *s2 );`	Ищет первое вхождение строки `s2` в строке `s1`. В случае успеха возвращает указатель на символ на начало строки `s2` в строке `s1`. В противном случае возвращает `NULL`
`char *strtok( char *s1, const char *s2 );`	Последовательность вызовов функции `strtok` позволяет разбить строку `s1` на лексемы – логические фрагменты текста, такие как слова, – отделяемые друг от друга символами, содержащимися в строке `s2`. В первый вызов в первом аргументе передается начальная строка `s1`, а в последующие, если продолжается разбивка на лексемы той же самой строки, – значение `NULL`. Каждый вызов возвращает указатель на текущую лексему. После возврата последней лексемы следующий вызов `strtok` вернет `NULL`

8.8.1 Функция strchr

Функция `strchr` ищет *первое вхождение* указанного символа в строке. В случае успеха `strchr` возвращает указатель на этот символ; в противном случае возвращается `NULL`. Программа в примере 8.14 ищет первое вхождение символов `'a'` и `'z'` в строке `"This is a test"`.

Пример 8.14 | Демонстрация функции strchr

```
// Пример 8.14: fig08_20.c
// Демонстрация функции strchr
#include <stdio.h>
#include <string.h>

int main( void )
{
   const char *string = "This is a test"; // инициализировать указатель
   char character1 = 'a'; // инициализировать character1
   char character2 = 'z'; // инициализировать character2

   // если символ character1 найден в строке
   if ( strchr( string, character1 ) != NULL ) {
      printf( "\'%c\' was found in \"%s\".\n",
         character1, string );
   } // конец if
   else { // если символ character1 найден
      printf( "\'%c\' was not found in \"%s\".\n",
         character1, string );
   } // конец else

   // если символ character2 найден в строке
   if ( strchr( string, character2 ) != NULL ) {
      printf( "\'%c\' was found in \"%s\".\n",
         character2, string );
   } // конец if
   else { // если символ character2 найден
      printf( "\'%c\' was not found in \"%s\".\n",
         character2, string );
   } // конец else
} // конец main
```

```
'a' was found in "This is a test".
'z' was not found in "This is a test".
```

8.8.2 Функция strcspn

Функция `strcspn` (пример 8.15) определяет длину начального фрагмента строки в первом аргументе, не содержащего символы из строки во втором аргументе. Возвращает длину найденного фрагмента.

Пример 8.15 | Демонстрация функции strcspn

```
// Пример 8.15: fig08_21.c
// Демонстрация функции strcspn
#include <stdio.h>
#include <string.h>
```

```
 5
 6 int main( void )
 7 {
 8    // инициализировать два указателя
 9    const char *string1 = "The value is 3.14159";
10    const char *string2 = "1234567890";
11
12    printf( "%s%s\n%s%s\n\n%s\n%s%u\n",
13       "string1 = ", string1, "string2 = ", string2,
14       "The length of the initial segment of string1",
15       "containing no characters from string2 = ",
16       strcspn( string1, string2 ) );
17 } // конец main
```

```
string1 = The value is 3.14159
string2 = 1234567890

The length of the initial segment of string1
containing no characters from string2 = 13
```

8.8.3 Функция strpbrk

Функция `strpbrk` ищет в строке, передаваемой в первом аргументе, *первое вхождение* любого символа из строки во втором аргументе. В случае успеха возвращает указатель на найденный символ в первой строке; в противном случае возвращает `NULL`. Программа в примере 8.16 ищет в строке `string1` первое вхождение любого символа из строки `string2`.

Пример 8.16 | Демонстрация функции strpbrk

```
 1 // Пример 8.16: fig08_22.c
 2 // Демонстрация функции strpbrk
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 int main( void )
 7 {
 8    const char *string1 = "This is a test"; // инициализировать указатель
 9    const char *string2 = "beware";         // инициализировать указатель
10
11    printf( "%s\"%s\"\n'%c'%s\n\"%s\"\n",
12       "Of the characters in ", string2,
13       *strpbrk( string1, string2 ),
14       " appears earliest in ", string1 );
15 } // конец main
```

```
Of the characters in "beware"
'a' appears earliest in
"This is a test"
```

8.8.4 Функция strrchr

Функция `strrchr` ищет *последнее вхождение* указанного символа в строке. В случае успеха возвращает указатель на этот символ; в противном случае

возвращается NULL. Программа в примере 8.17 ищет первое вхождение символа 'z' в строке "A zoo has many animals including zebras".

Пример 8.17 | Демонстрация функции strrchr

```
// Пример 8.17: fig08_23.c
// Демонстрация функции strrchr
#include <stdio.h>
#include <string.h>

int main( void )
{
   // инициализировать указатель
   const char *string1 = "A zoo has many animals including zebras";

   int c = 'z'; // искомый символ

   printf( "%s\n%s'%c'%s\"%s\"\n",
      "The remainder of string1 beginning with the",
      "last occurrence of character ", c,
      " is: ", strrchr( string1, c ) );
} // конец main
```

```
The remainder of string1 beginning with the
last occurrence of character 'z' is: "zebras"
```

8.8.5 Функция strspn

Функция strspn (пример 8.18) определяет длину начального фрагмента строки в первом аргументе, содержащего только символы из строки во втором аргументе. Возвращает длину найденного фрагмента.

Пример 8.18 | Демонстрация функции strspn

```
// Пример 8.18: fig08_24.c
// Демонстрация функции strspn
#include <stdio.h>
#include <string.h>

int main( void )
{
   // инициализировать два указателя
   const char *string1 = "The value is 3.14159";
   const char *string2 = "aehi lsTuv";

   printf( "%s%s\n%s%s\n\n%s\n%s%u\n",
      "string1 = ", string1, "string2 = ", string2,
      "The length of the initial segment of string1",
      "containing only characters from string2 = ",
      strspn( string1, string2 ) );
} // конец main
```

```
string1 = The value is 3.14159
string2 = aehi lsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

8.8.6 Функция strstr

Функция `strstr` ищет в строке в первом аргументе *первое вхождение* строки во втором аргументе. Если вторая строка будет найдена в первой строке, `strstr` вернет указатель на ее вхождение. Программа в примере 8.19 использует функцию `strstr`, чтобы найти строку `"def"` в строке `"abcdefabcdef"`.

Пример 8.19 | Демонстрация функции strstr

```
// Пример 8.19: fig08_25.c
// Демонстрация функции strstr
#include <stdio.h>
#include <string.h>

int main( void )
{
   const char *string1 = "abcdefabcdef"; // строка, где выполняется поиск
   const char *string2 = "def";          // искомая строка

   printf( "%s%s\n%s%s\n\n%s\n%s%s\n",
      "string1 = ", string1, "string2 = ", string2,
      "The remainder of string1 beginning with the",
      "first occurrence of string2 is: ",
      strstr( string1, string2 ) );
} // конец main
```

```
string1 = abcdefabcdef
string2 = def

The remainder of string1 beginning with the
first occurrence of string2 is: defabcdef
```

8.8.7 Функция strtok

Функция `strtok` (пример 8.20) используется для разбиения строки на последовательность **лексем**. Лексема – это последовательность символов между **разделителями** (обычно *пробелами* или *знаками пунктуации*, но вообще разделителем может быть *любой символ*). Например, в строке текста отдельные слова можно считать лексемами, а пробелы и знаки пунктуации – разделителями.

Пример 8.20 | Демонстрация функции strtok

```
// Пример 8.20: fig08_26.c
// Демонстрация функции strtok
#include <stdio.h>
#include <string.h>

int main( void )
{
   // инициализировать массив-строку
   char string[] = "This is a sentence with 7 tokens";
   char *tokenPtr; // создать указатель

```

```
12    printf( "%s\n%s\n\n%s\n",
13       "The string to be tokenized is:", string,
14       "The tokens are:" );
15
16   tokenPtr = strtok( string, " " ); // приступить к выделению лексем
17
18    // продолжать выделение лексем, пока tokenPtr не получит значение NULL
19    while ( tokenPtr != NULL ) {
20       printf( "%s\n", tokenPtr );
21       tokenPtr = strtok( NULL, " " ); // получить следующую лексему
22    } // конец while
23 } // конец main
```

```
The string to be tokenized is:
This is a sentence with 7 tokens
The tokens are:
This
is
a
sentence
with
7
tokens
```

Чтобы *разбить строку на лексемы*, необходимо выполнить несколько вызовов `strtok` (если предположить, что строка содержит более одной лексемы). В первый вызов `strtok` передаются два аргумента: строка для разбиения и строка с символами-разделителями. Инструкция в строке 16

```
16   tokenPtr = strtok( string, " " ); // приступить к выделению лексем
```

присвоит переменной `tokenPtr` указатель на первую лексему в строке. Второй аргумент, `" "`, указывает, что лексемы в исходной строке отделяются друг от друга пробелами. Функция `strtok` ищет в строке `string` первый символ, не являющийся разделителем (пробелом). Этот символ является началом первой лексемы. Затем функция ищет следующий символ-разделитель и *замещает его нулевым символом* (`'\0'`), чтобы завершить текущую лексему. Функция `strtok` сохраняет указатель на следующий символ, идущий в строке за текущей лексемой, и возвращает указатель на текущую лексему.

Последующие вызовы `strtok` в строке 21 будут продолжать возвращать очередные лексемы. В эти вызовы *в первом аргументе передается значение* `NULL`. Значение `NULL` в первом аргументе сообщает функции, что она должна продолжать выделять лексемы с позиции в строке, сохраненной последним вызовом `strtok`. Если все лексемы уже были возвращены предыдущими вызовами, `strtok` вернет `NULL`. В каждом новом вызове `strtok` допускается передавать новую строку с разделителями. Программа в примере 8.20 использует функцию `strtok` для выделения лексем из строки `"This is a sentence with 7 tokens"`. Каждую лексему программа выводит в отдельной строке. Функция `strtok` *изменяет исходную строку*, помещая в нее символ `'\0'` в конец каждой найденной лексемы; поэтому, если строка еще будет ис-

пользоваться далее в программе после вызова `strtok`, необходимо создать ее копию. [*Обратите внимание*: ознакомьтесь также с рекомендацией STR06-C центра CERT.]

8.9 Функции для работы с памятью

Библиотека функций для работы со строками, с которой мы продолжим знакомство в этом разделе, содержит также функции для выполнения различных операций с блоками памяти. Эти функции интерпретируют блоки памяти как массивы символов и способны оперировать любыми блоками данных. Некоторые из функций для работы с памятью перечислены в табл. 8.7. Термин «объект», используемый в обсуждении ниже, используется для обозначения блока данных. [*Обратите внимание*: для всех этих функций имеются более безопасные версии, определяемые приложением Annex K к стандарту C11. Мы представим их в разделе «Безопасное программирование на C» в конце этой главы, а также в приложении E.]

Таблица 8.7 | Функции для работы с памятью

Прототип	Описание
`void *memcpy( void *s1, const void *s2, size_t n );`	Копирует `n` символов из объекта `s2` в объект `s1`. Возвращает указатель на объект `s1`
`void *memmove( void *s1, const void *s2, size_t n );`	Копирует `n` символов из объекта `s2` в объект `s1`. Копирование выполняется, как если бы символы из объекта `s2` были сначала скопированы во временный массив, а затем из временного массива в объект `s1`. Возвращает указатель на объект `s1`
`int memcmp( const void *s1, const void *s2, size_t n );`	Сравнивает первые `n` символов объектов `s1` и `s2`. Функция возвращает 0, значение меньше или больше 0, если объект `s1` равен, меньше или больше объекта `s2` соответственно
`void *memchr( const void *s, int c, size_t n );`	Ищет первое вхождение символа `c` (как значения типа `unsigned char`) среди первых `n` символов в объекте `s`. В случае успеха возвращает указатель на символ `c`. В противном случае возвращает `NULL`
`void *memset( void *s, int c, size_t n );`	Копирует символ `c` (как значения типа `unsigned char`) в первые `n` символов объекта `s`. Возвращает указатель на объект `s`

Параметры-указатели объявлены с типом `void *`, поэтому они могут использоваться для выполнения операций с любыми данными в памяти. В главе 7 мы видели, что указателю типа `void *` можно присвоить значение

указателя любого типа, и значение указателя типа `void *` можно присвоить любому указателю. Благодаря этому данные функции могут принимать указатели на данные любого типа. Так как указатель типа `void *` не может быть разыменован, каждая функция принимает также параметр, определяющий размер блока памяти в символах (байтах), который должен быть обработан. Для простоты примеры в этом разделе оперируют массивами (блоками) символов. Функции в табл. 8.7 *не* проверяют наличие завершающего пустого символа.

8.9.1 Функция memcpy

Функция `memcpy` копирует указанное количество символов из объекта во втором аргументе в объект в первом аргументе. Функция может принимать указатели на объекты любых типов. Результат выполнения этой функции *не определен*, если объекты хранятся в памяти с перекрытием (то есть имеют общие части), – в таких случаях следует использовать функцию `memmove`. Программа в примере 8.21 использует `memcpy` для копирования строки из массива `s2` в массив `s1`.

Функция `memcpy` *более эффективна, чем* `strcpy`*, поэтому, когда количество копируемых символов известно, предпочтительнее использовать* `memcpy`*.*

Пример 8.21 | Демонстрация функции memcpy

```
 1 // Пример 8.21: fig08_28.c
 2 // Демонстрация функции memcpy
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 int main( void )
 7 {
 8    char s1[ 17 ]; // создать массив символов s1
 9    char s2[] = "Copy this string"; // инициализировать массив символов s2
10
11    memcpy( s1, s2, 17 );
12    printf( "%s\n%s\"%s\"\n",
13       "After s2 is copied into s1 with memcpy,",
14       "s1 contains ", s1 );
15 } // конец main
```

```
After s2 is copied into s1 with memcpy,
s1 contains "Copy this string"
```

8.9.2 Функция memmove

Функция `memmove` действует подобно функции `memcpy`. Она *копирует указанное количество байтов* из объекта во втором аргументе в объект в первом аргументе. Копирование выполняется, как если бы байты из объекта во втором аргументе копировались сначала во временный массив, а затем из времен-

ного массива в блок в первом аргументе. Это позволяет копировать *перекрывающиеся* объекты. Программа в примере 8.22 с помощью `memmove` копирует последние 10 байт из массива `x` в первые 10 байт того же массива `x`.

Результат выполнения функций для работы со строками, кроме `memmove`*, которые копируют символы, не определен, когда копирование выполняется в пределах одной и той же строки.*

Пример 8.22 | Демонстрация функции memmove

```
 1 // Пример 8.22: fig08_29.c
 2 // Демонстрация функции memmove
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 int main( void )
 7 {
 8    char x[] = "Home Sweet Home"; // инициализировать массив символов x
 9
10    printf( "%s%s\n", "The string in array x before memmove is: ", x );
11    printf( "%s%s\n", "The string in array x after memmove is: ",
12       (char *) memmove( x, &x[ 5 ], 10 ) );
13 } // конец main
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is: Sweet Home Home
```

8.9.3 Функция memcmp

Функция `memcmp` (пример 8.23) сравнивает указанное количество символов в объекте в первом аргументе с соответствующими символами в объекте во втором аргументе. Она возвращает 0, если объекты равны, *отрицательное значение* – если первый объект меньше второго, и *положительное значение* – если больше.

Пример 8.23 | Демонстрация функции memcmp

```
 1 // Пример 8.23: fig08_30.c
 2 // Демонстрация функции memcmp
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 int main( void )
 7 {
 8    char s1[] = "ABCDEFG"; // инициализировать массив символов s1
 9    char s2[] = "ABCDXYZ"; // инициализировать массив символов s2
10
11    printf( "%s%s\n%s%s\n\n%s%2d\n%s%2d\n%s%2d\n",
12       "s1 = ", s1, "s2 = ", s2,
13       "memcmp( s1, s2, 4 ) = ", memcmp( s1, s2, 4 ),
14       "memcmp( s1, s2, 7 ) = ", memcmp( s1, s2, 7 ),
15       "memcmp( s2, s1, 7 ) = ", memcmp( s2, s1, 7 ) );
16 } // конец main
```

```
s1 = ABCDEFG
s2 = ABCDXYZ

memcmp( s1, s2, 4 ) =  0
memcmp( s1, s2, 7 ) = -1
memcmp( s2, s1, 7 ) =  1
```

8.9.4 Функция memchr

Функция `memchr` ищет *первое вхождение байта*, представленного значением типа `unsigned char`, среди указанного количества байтов объекта. Если искомый байт будет найден, функция вернет указатель на него; в противном случае она вернет `NULL`. Программа в примере 8.24 ищет символ (байт) `'r'` в строке `"This is a string"`.

Пример 8.24 | Демонстрация функции memchr

```
 1 // Пример 8.24: fig08_31.c
 2 // Демонстрация функции memchr
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 int main( void )
 7 {
 8    const char *s = "This is a string"; // инициализировать указатель
 9
10    printf( "%s\'%c\'%s\"%s\"\n",
11       "The remainder of s after character ", 'r',
12       " is found is ", (char *) memchr( s, 'r', 16 ) );
13 } // конец main
```

```
The remainder of s after character 'r' is found is "ring"
```

8.9.5 Функция memset

Функция `memset` копирует значение байта во втором аргументе в первые `n` байт объекта в первом аргументе, где `n` определяется третьим аргументом. Программа в примере 8.25 использует функцию `memset`, чтобы скопировать `'b'` в первые 7 байт строки `string1`.

Используйте `memset` для заполнения массивов нулями вместо обхода элементов массива в цикле и присваивания им значений 0 по отдельности. Так, в примере 6.1 массив из 10 элементов можно было бы инициализировать вызовом `memset(n, 0, 10);`. Многие аппаратные архитектуры имеют блочные инструкции копирования или очистки памяти, которые могут использоваться компилятором для оптимизации `memset` и достижения максимальной скорости очистки памяти.

Пример 8.25 | Демонстрация функции memset

```
1 // Пример 8.25: fig08_32.c
2 // Демонстрация функции memset
```

```
 3 #include <stdio.h>
 4 #include <string.h>
 5
 6 int main( void )
 7 {
 8    char string1[ 15 ] = "BBBBBBBBBBBBBB"; // инициализировать string1
 9
10    printf( "string1 = %s\n", string1 );
11    printf( "string1 after memset = %s\n",
12       (char *) memset( string1, 'b', 7 ) );
13 } // конец main
```

```
string1 = BBBBBBBBBBBBBB
string1 after memset = bbbbbbbBBBBBBB
```

8.10 Прочие функции для работы со строками

В библиотеке функций для работы со строками имеются еще две функции: `strerror` и `strlen`. Их краткое описание приводится в табл. 8.8.

Таблица 8.8 | Остальные функции для работы со строками

Прототип	Описание
`char *strerror( int errornum );`	Отображает код ошибки `errornum` в текстовое ее описание с учетом национальных настроек системы и особенностей компилятора (например, сообщения могут выводиться на разных языках, исходя из национальных настроек). Возвращает указатель на строку
`size_t strlen( const char *s );`	Определяет длину строки `s`. Возвращает количество символов, предшествующих завершающему нулевому символу

8.10.1 Функция strerror

Функция `strerror` принимает числовой код ошибки и возвращает указатель на строку с текстом сообщения, соответствующим этому коду. Программа в примере 8.26 демонстрирует применение функции `strerror`.

Пример 8.26 | Демонстрация функции strerror

```
1 // Пример 8.26: fig08_34.c
2 // Демонстрация функции strerror
3 #include <stdio.h>
4 #include <string.h>
5
6 int main( void )
7 {
8    printf( "%s\n", strerror( 2 ) );
9 } // конец main
```

```
No such file or directory
```

8.10.2 Функция strlen

Функция `strlen` принимает строку и возвращает количество символов в строке – завершающий нулевой символ не включается в это число. Программа в примере 8.27 демонстрирует применение функции `strlen`.

Пример 8.27 | Демонстрация функции strlen

```
// Пример 8.27: fig08_35.c
// Демонстрация функции strlen
#include <stdio.h>
#include <string.h>

int main( void )
{
   // инициализировать три указателя
   const char *string1 = "abcdefghijklmnopqrstuvwxyz";
   const char *string2 = "four";
   const char *string3 = "Boston";

   printf("%s\"%s\"%s%u\n%s\"%s\"%s%u\n%s\"%s\"%s%u\n",
      "The length of ", string1, " is ", strlen( string1 ),
      "The length of ", string2, " is ", strlen( string2 ),
      "The length of ", string3, " is ", strlen( string1 ) );
} // конец main
```

```
The length of "abcdefghijklmnopqrstuvwxyz" is 26
The length of "four" is 4
The length of "Boston" is 6
```

8.11 Безопасное программирование на C

Безопасные функции для работы со строками

В этой главе были представлены функции `sprintf`, `strcpy`, `strncpy`, `strcat`, `strncat`, `strtok`, `strlen`, `memcpy`, `memmove` и `memset`. В приложении Annex K к стандарту C11 определяются более безопасные версии этих и многих других функций для работы со строками. Если ваш компилятор соответствует положениям приложения Annex K, старайтесь использовать безопасные версии этих функций. Кроме всего прочего, безопасные версии помогают предотвратить ошибку переполнения буфера, принимая дополнительный параметр, который определяет количество элементов в целевом массиве, и проверяя указатели на равенство значению `NULL`.

Чтение чисел и проверка корректности ввода

Обязательно следует проверять все данные, которые вводятся в программе. Например, когда пользователю предлагается ввести число в диапазоне 1–100 и затем выполняется попытка ввести целое число с помощью функции `scanf`, возникает несколько потенциальных проблем. Пользователь может ввести целое число за пределами указанного диапазона, целое число за

пределами диапазона представления целых чисел на данном компьютере, нецелое числовое значение или вообще нечисловое значение.

С помощью функций, представленных в этой главе, можно реализовать исчерпывающую проверку ввода пользователя. Например, можно:

- с помощью `fgets` прочитать ввод в виде строки;
- преобразовать строку в число с помощью `strtol` и проверить успех преобразования;
- проверить попадание введенного значения в требуемый диапазон.

Дополнительную информацию и примеры приемов преобразования ввода в числовые значения можно найти в рекомендации INT05-C центра CERT по адресу: `www.securecoding.cert.org`.

9

Форматированный ввод/вывод

В этой главе вы:

- научитесь использовать потоки ввода и вывода;
- познакомитесь с широкими возможностями поддержки форматированного ввода и вывода;
- узнаете, как выводить информацию в поля требуемой ширины и с требуемой точностью;
- познакомитесь с флагами, используемыми в строке формата функции `printf`;
- узнаете, как организовать вывод литералов и экранированных последовательностей;
- научитесь вводить данные с помощью функции `scanf`.

9.1 Введение

Важной частью решения любой задачи является *представление* результатов. В этой главе мы обсудим во всех подробностях возможности механизма форматирования, реализованного в виде функций `scanf` и `printf`. Эти функции вводят данные из **потока стандартного ввода** и выводят данные в **поток стандартного вывода**. Не забудьте подключить заголовочный файл `<stdio.h>` в программе, если собираетесь использовать эти функции. В главе 11 обсуждаются некоторые дополнительные функции, включая функции из стандартной библиотеки ввода/вывода (`<stdio.h>`).

9.2 Потоки данных

Весь ввод и вывод осуществляется с помощью **потоков данных**, которые являются последовательностями байтов. В операциях *ввода* потоки байтов текут *из устройств* (таких как клавиатура, жесткий диск, сетевая карта) *в оперативную память*. В операциях вывода потоки байтов текут в обратном направлении, *из оперативной памяти в устройства* (например, на экран компьютера, в принтер, на жесткий диск, в сетевую карту и т. д.).

Когда программа только начинает выполнение, она автоматически подключается к трем потокам данных. *Поток стандартного ввода* обычно связан с *клавиатурой*, а *поток стандартного вывода* – с *экраном*. Операционные системы позволяют подключать эти потоки к другим устройствам. Третий поток данных, **стандартный поток вывода ошибок**, подключен к *экрану*. Порядок вывода сообщений об ошибках в *стандартный поток вывода ошибок* будет описан в главе 11, где мы также во всех деталях обсудим потоки данных.

9.3 Форматированный вывод с помощью printf

Форматирование вывода в программах часто выполняется с помощью функции `printf`. В каждый вызов функции `printf` передается строка, описывающая формат вывода. **Строка формата** может содержать **спецификаторы преобразования**, **флаги**, описания **ширины полей вывода**, описание **точности представления** и **литеральные символы**. Вместе со знаком процента (`%`) они образуют **спецификаторы преобразования**. Функция `printf` обладает следующими возможностями форматирования, каждая из которых обсуждается в этой главе:

1. Округление вещественных значений до указанного знака после десятичной точки.
2. Выравнивание колонок вещественных чисел так, чтобы десятичные точки отображались друг под другом.
3. Выравнивание вывода по левому или правому краю.
4. Вставка литеральных символов в точную позицию в строке.
5. Представление вещественных чисел в экспоненциальном формате.
6. Представление беззнаковых целых чисел в восьмеричной или шестнадцатеричной системе счисления. Подробно восьмеричные и шестнадцатеричные числа описываются в приложении С.
7. Вывод данных любых типов в поля фиксированной ширины и с заданной точностью.

Функция `printf` имеет следующий синтаксис:

```
printf( строка формата, остальные аргументы );
```

Строка формата описывает формат вывода, а в *остальных аргументах* (которые являются необязательными) передаются значения для спецификаторов преобразования, присутствующих в *строке формата*. Каждый спецификатор формата начинается со знака процента и заканчивается символом спецификатора. В одной строке формата может быть сколько угодно много спецификаторов формата.

Часто программисты забывают закрыть строку формата двойной кавычкой, что является синтаксической ошибкой.

9.4 Вывод целых чисел

Целые числа – это числа, не имеющие дробной части, такие как 776, 0 или –52. Вывод целочисленных значений может осуществляться с помощью множества спецификаторов формата, которые перечисляются в табл. 9.1.

Таблица 9.1 | Спецификаторы преобразования целых чисел

Спецификатор	Описание
`d`	Выводит числовое значение как целое десятичное число со знаком
`i`	Выводит числовое значение как целое десятичное число со знаком. [*Обратите внимание*: спецификаторы `i` и `d` отличаются при использовании в функциях `printf` и `scanf`]
`o`	Выводит числовое значение как восьмеричное целое число без знака
`u`	Выводит числовое значение как десятичное целое число без знака
`x` или `X`	Выводит числовое значение как шестнадцатеричное целое число без знака. Спецификатор `X` выводит буквы верхнего регистра A–F, а спецификатор `x` – буквы нижнего регистра a–f
`h`, `l` или `ll`	Помещаются перед другими спецификаторами преобразования целых чисел, чтобы обозначить короткое (`short`), длинное (`long`) или очень длинное (`long long`) целочисленное значение соответственно. Их часто называют модификаторами длины

Программа в примере 9.1 выводит целые числа, используя разные спецификаторы. Обычно для чисел выводится только знак «минус», а знак «плюс» не выводится. Далее в этой главе будет показано, как принудительно вывести знак «плюс». Кроме того, отрицательное значение `-455` при выводе с помощью спецификатора `%u` (строка 15) интерпретируется как беззнаковое значение `4294966841`.

Вывод отрицательных значений с помощью спецификатора вывода беззнаковых значений является ошибкой.

Пример 9.1 | Демонстрация спецификаторов преобразования целых чисел

```
 1 // Пример 9.1: fig09_02.c
 2 // Демонстрация спецификаторов преобразования целых чисел
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    printf( "%d\n", 455 );
 8    printf( "%i\n", 455 );          // в printf i действует так же, как и d
 9    printf( "%d\n", +455 );         // знак "плюс" не выводится
10    printf( "%d\n", -455 );         // выведет знак "минус"
11    printf( "%hd\n", 32000 );
12    printf( "%ld\n", 2000000000L ); // L превращает литерал в длинное целое
13    printf( "%o\n", 455 );          // восьмеричное
14    printf( "%u\n", 455 );
15    printf( "%u\n", -455 );
16    printf( "%x\n", 455 ); // шестнадцатеричное, a-f в нижнем регистре
17    printf( "%X\n", 455 ); // шестнадцатеричное, A-F в нижнем регистре
18 } // конец main
```

```
455
455
455
-455
32000
2000000000
707
455
4294966841
1c7
1C7
```

9.5 Вывод вещественных чисел

Вещественное значение содержит десятичную точку, как, например, в числах 33.5, 0.0 или –657.983. Вещественные значения могут отображаться в одном из нескольких форматов. В табл. 9.2 перечислены спецификаторы преобразования для вещественных чисел. Спецификаторы `e` и `E` отображают вещественные значения в **экспоненциальном формате** – компьютерном эквиваленте **научной формы записи**, используемой в математике. Например, значение 150.4582 в научной форме записи имеет вид:

```
1.504582 × 10²
```

а в экспоненциальной:

```
1.504582E+02
```

Такая форма записи сообщает, что число `1.504582` умножается на 10 во второй степени (`E+02`). Символ `E` происходит от слова «exponent» (экспонента).

Таблица 9.2 | Спецификаторы преобразования вещественных чисел

Спецификатор	Описание
`e` или `E`	Выводит вещественное значение в экспоненциальной форме записи
`f` или `F`	Выводит вещественное значение в форме записи с фиксированной десятичной точкой
`g` или `G`	Выводит вещественное значение либо в форме записи с фиксированной десятичной точкой, как спецификатор `f`, либо в экспоненциальной форме записи, как спецификатор `e` (или `E`), исходя из порядковой величины значения
`L`	Добавляется перед спецификаторами преобразования вещественных чисел, чтобы показать, что отображается значение типа `long double`

При выводе вещественных значений с применением спецификаторов `e`, `E` и `f` по умолчанию отображаются с точностью до *шести знаков* после десятичной точки (например, 1.04592); существует возможность явно

указать другую точность. Спецификатор преобразования `f` всегда выводит хотя бы один знак *слева* от десятичной точки. Спецификаторы `e` и `E` выводят букву `e` *нижнего регистра* и букву `E` *верхнего регистра* соответственно перед экспонентой и отображают *точно одну* цифру слева от десятичной точки.

Спецификатор преобразования `g` (или `G`) выводит значение либо в формате спецификатора `e` (`E`), либо в формате спецификатора `f` без завершающих нулей (то есть число `1.234000` выводится как `1.234`). Значение выводится в экспоненциальной форме, если после преобразования значение экспоненты меньше –4 или больше или равно значению указанной точности (по умолчанию *шесть значащих цифр* для `g` и `G`). В противном случае значение выводится в формате спецификатора `f`. Завершающие нули в дробной части при этом не отображаются. Для вывода десятичной точки требуется, чтобы отображался хотя бы один знак после нее. Значения `0.0000875`, `8750000.0`, `8.75` и `87.50` выводятся спецификатором `g` как `8.75e-05`, `8.75e+06`, `8.75` и `87.5`. Экспоненциальная форма записи используется для значения `0.0000875` потому, что после преобразования в экспоненциальную форму значение его экспоненты (`-5`) меньше `-4`, а для значения `8750000.0` потому, что его экспонента (`6`) равна точности по умолчанию.

Точность спецификатора `g` и `G` определяет максимальное количество значащих цифр для вывода, *включая* цифру *слева* от десятичной точки. Значение `1234567.0` будет выведено спецификатором `%g` как `1.23457e+06` (не забывайте, что для всех спецификаторов преобразования вещественных чисел *по умолчанию используется точность* 6). Здесь в результате имеется шесть значащих цифр. Разница между спецификаторами `g` и `G` в точности соответствует разнице между спецификаторами `e` и `E`, когда значение выводится в экспоненциальной форме, – спецификатор `g` выводит букву `e` нижнего регистра, а спецификатор `G` – букву `E` верхнего регистра.

При выводе данных убедитесь, что пользователь знает о возможном появлении неточности из-за округлений при форматированном выводе (например, при выводе с определенной точностью).

Программа в примере 9.2 демонстрирует применение всех спецификаторов преобразования вещественных значений. Спецификаторы `%E`, `%e` и `%g` вызывают *округление* значения при выводе, а спецификатор `%f` – нет. [*Обратите внимание*: в некоторых компиляторах при выводе экспоненты отображаются два знака справа от знака `+`.]

Пример 9.2 | Демонстрация спецификаторов преобразования вещественных чисел

```
1 // Пример 9.2: fig09_04.c
2 // Демонстрация спецификаторов преобразования вещественных чисел
3 #include <stdio.h>
4
```

```
 5 int main( void )
 6 {
 7    printf( "%e\n", 1234567.89 );
 8    printf( "%e\n", +1234567.89 ); // знак "плюс" не выводится
 9    printf( "%e\n", -1234567.89 ); // знак "минус" выводится
10    printf( "%E\n", 1234567.89 );
11    printf( "%f\n", 1234567.89 );
12    printf( "%g\n", 1234567.89 );
13    printf( "%G\n", 1234567.89 );
14 } // конец main
```

```
1.234568e+006
1.234568e+006
-1.234568e+006
1.234568E+006
1234567.890000
1.23457e+006
1.23457E+006
```

9.6 Вывод строк и символов

Спецификаторы c и s используются для вывода отдельных символов и строк соответственно. Спецификатор c требует, чтобы аргумент имел тип char. Спецификатор s требует, чтобы аргумент был указателем на значение типа char. Спецификатор продолжает вывод символов в строке, пока не будет встречен завершающий нулевой символ ('\0'). Программа в примере 9.3 демонстрирует вывод символов и строк с помощью спецификаторов c и s.

Пример 9.3 | Демонстрация спецификаторов преобразования символов и строк

```
 1 // Пример 9.3: fig09_05c
 2 // Демонстрация спецификаторов преобразования символов и строк
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    char character = 'A';               // инициализировать символ
 8    char string[] = "This is a string"; // инициализировать массив
                                          // символов
 9    const char *stringPtr = "This is also a string"; // указатель
10
11    printf("%c\n", character );
12    printf("%s\n", "This is a string" );
13    printf("%s\n", string );
14    printf("%s\n", stringPtr );
15 } // конец main
```

```
A
This is a string
This is a string
This is also a string
```

Использование спецификатора `%c` *для вывода строки является ошибкой. Спецификатор* `%c` *требует передачи аргумента типа* `char`*, тогда как строка является указателем на значение типа* `char` *(то есть имеет тип* `char *`*).*

Попытка использовать спецификатор `%s` *для вывода аргумента типа* `char` *часто приводит к аварийному завершению программы, вызванному нарушением прав доступа к памяти. Спецификатор* `%s` *требует передачи в аргументе указателя на значение типа* `char`*.*

Заключение строк символов в одиночные кавычки является синтаксической ошибкой. Строки символов должны заключаться в двойные кавычки.

Заключение литералов символов в двойные кавычки создает указатель на строку с двумя символами, вторым из которых является завершающий нулевой символ.

9.7 Прочие спецификаторы формата

В табл. 9.3 описываются спецификаторы `p` и `%`. Программа в примере 9.4 использует спецификатор `%p` для вывода значения указателя `ptr` и адреса переменной `x`; эти значения идентичны, потому что указателю `ptr` присваивается адрес `x`. Последняя инструкция `printf` использует спецификатор `%%` для вывода символа `%`.

Спецификатор преобразования `p` *выводит адрес в формате, зависящем от реализации (во многих системах вместо десятичной используется шестнадцатеричная форма записи).*

Попытка вывода знака процента `%` *как одиночного символа вместо последовательности* `%%` *в строке формата является ошибкой. Когда символ* `%` *находится в строке формата, ему должен предшествовать спецификатор* `%`*.*

Таблица 9.3 | Прочие спецификаторы преобразования

Спецификатор	Описание
`p`	Выводит значение указателя в формате, зависящем от реализации
`%`	Выводит символ `%`

Пример 9.4 | Демонстрация спецификаторов преобразования p и %

```
// Пример 9.4: fig09_07.c
// Демонстрация спецификаторов преобразования p и %
#include <stdio.h>

int main( void )
{
   int *ptr;      // определить указатель на значение типа int
   int x = 12345; // инициализировать переменную x

   ptr = &x; // присвоить указателю ptr адрес переменной x
   printf( "The value of ptr is %p\n", ptr );
   printf( "The address of x is %p\n\n", &x );

   puts( "Printing a %% in a format control string" );
} // конец main
```

```
The value of ptr is 002EF778
The address of x is 002EF778
Printing a % in a format control string
```

9.8 Вывод с указанием ширины поля и точности

Точный размер поля, куда выводятся данные, определяется параметром **ширины поля**. Если ширина поля больше, чем требуется для вывода данных, данные обычно *выравниваются по правому краю* поля. Целое число, определяющее ширину поля, вставляется между знаком процента (%) и спецификатором преобразования (например, %4d). Программа в примере 9.5 выводит две группы по пять чисел, количество цифр в которых меньше ширины поля, с *выравниванием по правому краю*. Если для вывода данных требуется больше места, ширина поля автоматически увеличивается. Имейте в виду, что знак «минус» в отрицательных значениях занимает одно знакоместо в поле вывода и должен учитываться при определении ширины поля. Значение ширины поля можно указывать во всех спецификаторах преобразования.

Определение недостаточно большой ширины поля может привести к смещению остальных данных в выводе, сделав его трудночитаемым. Изучайте свои данные!

Пример 9.5 | Демонстрация вывода целых чисел с выравниванием по правому краю

```
// Пример 9.5: fig09_08.c
// Демонстрация вывода целых чисел с выравниванием по правому краю
#include <stdio.h>

int main( void )
```

```
 6 {
 7    printf( "%4d\n", 1 );
 8    printf( "%4d\n", 12 );
 9    printf( "%4d\n", 123 );
10    printf( "%4d\n", 1234 );
11    printf( "%4d\n\n", 12345 );
12
13    printf( "%4d\n", -1 );
14    printf( "%4d\n", -12 );
15    printf( "%4d\n", -123 );
16    printf( "%4d\n", -1234 );
17    printf( "%4d\n", -12345 );
18 } // конец main
```

```
   1
  12
 123
1234
12345

  -1
 -12
-123
-1234
-12345
```

Функция `printf` позволяет также указать *точность* вывода данных. Понятие точности имеет разный смысл для данных разных типов. При использовании со спецификаторами преобразования целых чисел точность определяет *минимальное количество выводимых цифр*. Если фактическое значение содержит меньше цифр, чем определено значением точности, и определение точности включает ведущий ноль или десятичную точку, при выводе число дополняется нулями слева до достижения общего количества цифр в числе значения точности. Если определение точности не включает ни ноль, ни десятичную точку, дополнение выполняется пробелами. По умолчанию при выводе целых чисел используется точность, равная 1. При использовании со спецификаторами преобразования вещественных чисел – `e`, `E` и `f` – точность определяет *количество знаков после десятичной точки*. При использовании со спецификаторами `g` и `G` точность определяет *максимальное количество значащих разрядов для вывода*. При использовании со спецификатором `s` точность определяет *максимальное количество выводимых символов*.

Значение точности определяется как десятичная точка (.), за которой следует целое число, между знаком процента и спецификатором преобразования. Программа в примере 9.6 демонстрирует использование точности в строке формата. Если при выводе вещественных чисел значение точности оказывается меньше количества десятичных знаков в дробной части, выводимое значение округляется.

Пример 9.6 | Вывод целых и вещественных чисел, а также строк с указанной точностью

```
1 // Пример 9.6: fig09_09.c
2 // Вывод целых и вещественных чисел, а также строк с указанной точностью
3 #include <stdio.h>
4
5 int main( void )
6 {
7    int i = 873;                 // инициализировать int i
8    double f = 123.94536;        // инициализировать double f
9    char s[] = "Happy Birthday"; // инициализировать массив символов s
10
11   puts( "Using precision for integers" );
12   printf( "\t%.4d\n\t%.9d\n\n", i, i );
13
14   puts( "Using precision for floating-point numbers" );
15   printf( "\t%.3f\n\t%.3e\n\t%.3g\n\n", f, f, f );
16
17   puts( "Using precision for strings" );
18   printf( "\t%.11s\n", s );
19 } // конец main
```

```
Using precision for integers
        0873
        000000873

Using precision for floating-point numbers
        123.945
        1.239e+002
        124

Using precision for strings
        Happy Birth
```

Ширину поля и точность можно объединять, помещая между знаком процента и спецификатором значение ширины поля, затем десятичную точку и значение точности, как в следующей инструкции:

```
printf( "%9.3f", 123.456789 );
```

которая выведет `123.457` с тремя знаками после десятичной точки и с выравниванием по правому краю в поле шириной девять символов.

Ширину поля и точность можно также указывать динамически, с помощью дополнительных целочисленных аргументов, следующих за строкой формата. Для этого достаточно вставить символ звездочки (*) на место значения ширины поля или точности (или того и другого). При выполнении вместо звездочек будут использованы соответствующие целочисленные аргументы. Значение ширины поля может быть положительным или отрицательным (если указано отрицательное значение ширины поля, данные будут выводиться с выравниванием по левому краю, как описывается в следующем разделе). Инструкция

```
printf( "%*.*f", 7, 2, 98.736 );
```

выведет указанное значение как `98.74` с выравниванием по правому краю в поле шириной `7` символов и с точностью `2` знака после десятичной точки.

9.9 Использование флагов в строке формата функции printf

Функция `printf` поддерживает также дополнительные *флаги*, управляющие некоторыми особенностями форматирования вывода. В строке формата можно использовать пять флагов (табл. 9.4). Флаги должны следовать непосредственно за знаком процента. Допускается в одном спецификаторе преобразования использовать сразу несколько флагов.

Таблица 9.4 | Флаги в строке формата

Спецификатор	Описание
`-` (*знак минус*)	Вывод в указанном поле с выравниванием по левому краю
`+` (*знак плюс*)	Выводить знак «плюс» (`+`) перед положительными значениями и знак «минус» (`-`) перед отрицательными
пробел	Выводить пробел перед положительными значениями вместо знака «плюс» (`+`)
`#`	Добавлять ведущий `0` при выводе значения в восьмеричном представлении со спецификатором `o`. Добавлять префикс `0x` или `0X` при выводе значения в шестнадцатеричном представлении со спецификатором `x` или `X`. Принудительно добавлять десятичную точку при выводе вещественных значений со спецификатором `e`, `E`, `f`, `g` или `G`, не имеющим дробной части. (Обычно десятичная точка выводится, только если за ней следует хотя бы одна цифра.) Спецификаторы `g` и `G` не устраняют завершающие нули
`0` (*ноль*)	Дополнять число слева нулями до ширины поля

Программа в примере 9.7 демонстрирует вывод строки, целого и вещественного чисел, а также символа с выравниванием по левому и по правому краю.

Пример 9.7 | Вывод значений с выравниванием по левому и по правому краю

```
1 // Пример 9.7: fig09_11.c
2 // Вывод значений с выравниванием по левому и по правому краю
3 #include <stdio.h>
4
5 int main( void )
6 {
7    printf( "%10s%10d%10c%10f\n\n", "hello", 7, 'a', 1.23 );
8    printf( "%-10s%-10d%-10c%-10f\n", "hello", 7, 'a', 1.23 );
9 } // конец main
```

```
     hello         7         a  1.230000
hello    7          a         1.230000
```

Программа в примере 9.8 выводит положительное и отрицательное число, с флагом + и без него. Знак «минус» (-) выводится в обоих случаях, а знак «плюс» (+) – только при использовании флага +.

Пример 9.8 | Вывод положительных и отрицательных чисел с флагом + и без него

```
1 // Пример 9.8: fig09_12.c
2 // Вывод положительных и отрицательных чисел с флагом + и без него
3 #include <stdio.h>
4
5 int main( void )
6 {
7    printf( "%d\n%d\n", 786, -786 );
8    printf( "%+d\n%+d\n", 786, -786 );
9 } // конец main
```

```
786
-786
+786
-786
```

Программа в примере 9.9 предваряет положительные числа пробелом. Это может пригодиться для выравнивания положительных и одинаковых чисел с одинаковыми количествами цифр. Значение -547 не дополняется слева пробелом, потому что в нем присутствует знак «минус».

Пример 9.9 | Вывод положительных чисел с дополнительным пробелом впереди

```
1 // Пример 9.9: fig09_13.c
2 // Вывод положительных чисел с дополнительным пробелом впереди
3
4 #include <stdio.h>
5
6 int main( void )
7 {
8    printf( "% d\n% d\n", 547, -547 );
9 } // конец main
```

```
547
-547
```

Программа в примере 9.10 использует флаг # для добавления префикса 0 при выводе значений в восьмеричном представлении и префикса 0x или 0X в шестнадцатеричном представлении, а также для принудительного вывода десятичной точки при использовании спецификатора g.

Пример 9.10 | Использование флага # со спецификаторами преобразования o, x, X и любыми спецификаторами для вывода вещественных значений

```
1 // Пример 9.10: fig09_14.c
2 // Использование флага # со спецификаторами преобразования o, x, X
3 // и любыми спецификаторами для вывода вещественных значений
```

```
 4 #include <stdio.h>
 5
 6 int main( void )
 7 {
 8    int c = 1427;      // инициализировать c
 9    double p = 1427.0; // инициализировать p
10
11    printf( "%#o\n", c );
12    printf( "%#x\n", c );
13    printf( "%#X\n", c );
14    printf( "\n%g\n", p );
15    printf( "%#g\n", p );
16 } // конец main
```

```
02623
0x593
0X593

1427
1427.00
```

Программа в примере 9.11 объединяет флаги `+` и `0` (ноль) для вывода значения `452` в поле шириной 9 символов со знаком `+` и ведущими нулями, а затем выводит то же значение `452`, используя только флаг `0`.

Пример 9.11 | Демонстрация флага 0 (ноль)

```
1 // Пример 9.11: fig09_15.c
2 // Демонстрация флага 0 (ноль)
3 #include <stdio.h>
4
5 int main( void )
6 {
7    printf( "%+09d\n", 452 );
8    printf( "%09d\n", 452 );
9 } // конец main
```

```
+00000452
000000452
```

9.10 Вывод литералов и экранированных последовательностей

Большинство литералов символов при выводе с помощью `printf` можно просто включить в строку формата. Однако существуют некоторые «проблемные» символы, такие как *двойные кавычки* ("), которые сами используются для ограничения строки формата. Различные управляющие символы, такие как *перевод строки* и *табуляция*, должны быть представлены экранированными последовательностями (escape sequences). Экранированная последовательность начинается с символа обратного слэша (\), за которым следует экранируемый символ. В табл. 9.5 перечислены наиболее часто используемые экранированные последовательности и описано их назначение.

Таблица 9.5 | Экранированные последовательности

Экранированная последовательность	Описание
`\'` (*одиночная кавычка*)	Выводит символ одиночной кавычки (')
`\"` (*двойная кавычка*)	Выводит символ двойной кавычки (")
`\?` (*знак вопроса*)	Выводит символ знака вопроса (?)
`\\` (*обратный слэш*)	Выводит символ обратного слэша (\)
`\a` (*звонок или предупреждение*)	Вызывает звуковой или визуальный сигнал
`\b` (*забой*)	Перемещает текстовый курсор на одну позицию влево
`\f` (*перевод формата*)	Перемещает текстовый курсор в начало следующей логической страницы
`\n` (*перевод строки*)	Перемещает текстовый курсор в начало следующей строки
`\r` (*возврат каретки*)	Перемещает текстовый курсор в начало текущей строки
`\t` (*горизонтальная табуляция*)	Перемещает текстовый курсор в следующую позицию горизонтальной табуляции
`\v` (*вертикальная табуляция*)	Перемещает текстовый курсор в следующую позицию вертикальной табуляции

9.11 Чтение форматированного ввода с помощью функции scanf

Точный *формат ввода* можно описать с помощью функции `scanf`. Каждый вызов `scanf` содержит строку формата, описывающую формат входных данных. Строка формата состоит из спецификаторов преобразования и литералов символов. Функция `scanf` обладает следующими возможностями:

1) ввод данных любых типов;
2) ввод отдельных конкретных символов из потока ввода;
3) пропуск отдельных конкретных символов, присутствующих в потоке ввода.

Функция `scanf` имеет следующий синтаксис:

```
scanf( строка формата, другие аргументы );
```

где *строка формата* описывает формат входных данных, а *другие аргументы* являются указателями на переменные, в которых должны сохраняться введенные данные.

В каждом приглашении к вводу предлагайте пользователю ввести только один элемент данных. Избегайте приглашений, предлагающих ввести сразу несколько элементов.

Всегда предусматривайте действия в ответ на некорректный ввод, например когда пользователь введет целое число, бессмысленное в контексте программы, или строку, в которой отсутствуют знаки препинания или пробелы.

В табл. 9.6 перечислены спецификаторы преобразования для ввода данных любых типов. В оставшейся части этого раздела вы найдете программы, демонстрирующие чтение данных с применением функции `scanf` и различных спецификаторов преобразования.

Таблица 9.6 | Спецификаторы преобразования для функции scanf

Спецификатор	Описание
Целые числа	
`d`	Читает целое число, возможно со знаком. Соответствующий аргумент должен быть указателем на переменную типа `int`
`i`	Читает целое число, возможно со знаком, в десятичной, восьмеричной или шестнадцатеричной форме записи. Соответствующий аргумент должен быть указателем на переменную типа `int`
`o`	Читает целое число в восьмеричной форме записи. Соответствующий аргумент должен быть указателем на переменную типа `unsigned int`
`u`	Читает целое число без знака в десятичной форме записи. Соответствующий аргумент должен быть указателем на переменную типа `unsigned int`
`x` или `X`	Читает целое число в шестнадцатеричной форме записи. Соответствующий аргумент должен быть указателем на переменную типа `unsigned int`
`h`, `l` и `ll`	Помещаются перед другими спецификаторами преобразования целых чисел, чтобы обозначить короткое (`short`), длинное (`long`) или очень длинное (`long long`) целочисленное значение соответственно
Вещественные числа	
`e`, `E`, `f`, `g` или `G`	Читает вещественное число. Соответствующий аргумент должен быть указателем на переменную вещественного типа
`l` или `L`	Добавляется перед спецификаторами преобразования вещественных чисел, чтобы показать, что вводится значение типа `double` или `long double`. Соответствующий аргумент должен быть указателем на переменную типа `double` или `long double`
Строки символов	
`c`	Читает символ. Соответствующий аргумент должен быть указателем на переменную типа `char`; завершающий нулевой символ (`'\0'`) не добавляется
`s`	Читает строку. Соответствующий аргумент должен быть указателем на массив символов, достаточно длинный, чтобы вместить введенную строку и завершающий нулевой символ (`'\0'`), который добавляется автоматически

Таблица 9.6 | Спецификаторы преобразования для функции scanf

Спецификатор	Описание
Множество символов для сканирования	
[сканируемые символы]	Выполняет сканирование введенной строки на наличие перечисленных символов и сохраняет эти символы в массиве
Прочие	
`p`	Читает адрес, который должен быть оформлен так же, как и при выводе функцией `printf` с применением спецификатора `%p`
`n`	Сохраняет количество символов, введенных к данному моменту функцией `scanf`. Соответствующий аргумент должен быть указателем на переменную типа `int`
`%`	Пропускает знак процента (`%`) в введенной последовательности

Программа в примере 9.12 читает целые числа с применением различных спецификаторов преобразования целых чисел и выводит их в десятичном представлении. Спецификатор преобразования `%i` может вводить целые числа в десятичном, восьмеричном и шестнадцатеричном представлениях.

Пример 9.12 | Чтение ввода с применением спецификаторов преобразования целых чисел

```
 1 // Пример 9.12: fig09_18.c
 2 // Чтение ввода с применением спецификаторов преобразования целых чисел
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    int a;
 8    int b;
 9    int c;
10    int d;
11    int e;
12    int f;
13    int g;
14
15    puts( "Enter seven integers: " );
16    scanf( "%d%i%i%i%o%u%x", &a, &b, &c, &d, &e, &f, &g );
17
18    puts( "\nThe input displayed as decimal integers is:" );
19    printf( "%d %d %d %d %d %d %d\n", a, b, c, d, e, f, g );
20 } // конец main
```

```
Enter seven integers:
-70 -70 070 0x70 70 70 70

The input displayed as decimal integers is:
-70 -70 56 112 56 70 112
```

Для ввода вещественных чисел можно использовать любые спецификаторы преобразований: `e`, `E`, `f`, `g` и `G`. Программа в примере 9.13 читает

вещественные числа с применением всех трех возможных типов спецификаторов преобразования вещественных чисел и отображает их с помощью спецификатора преобразования `f`. Вывод программы подтверждает неточность представления вещественных чисел – особенно ярко это проявляется в третьем значении.

Пример 9.13 | Чтение ввода с применением спецификаторов преобразования вещественных чисел

```
// Пример 9.13: fig09_19.c
// Чтение ввода с применением спецификаторов преобразования
// вещественных чисел
#include <stdio.h>

// выполнение программы начинается с функции main
int main( void )
{
   double a;
   double b;
   double c;

   puts( "Enter three floating-point numbers:" );
   scanf( "%le%lf%lg", &a, &b, &c );

   puts( "\nHere are the numbers entered in plain:" );
   puts( "floating-point notation:\n" );
   printf( "%f\n%f\n%f\n", a, b, c );
} // конец main
```

```
Enter three floating-point numbers:
1.27987 1.27987e+03 3.38476e-06

Here are the numbers entered in plain
floating-point notation:
1.279870
1279.870000
0.000003
```

Символы и строки вводятся с применением спецификаторов преобразования `c` и `s` соответственно. Программа в примере 9.14 предлагает пользователю ввести строку. Она вводит первый символ строки с помощью спецификатора `%c` и сохраняет его в переменной `x`, а затем вводит остаток строки с помощью спецификатора `%s` и сохраняет в массиве символов `y`.

Пример 9.14 | Чтение символов и строк

```
// Пример 9.14: fig09_20.c
// Чтение символов и строк
#include <stdio.h>

int main( void )
{
   char x;
   char y[ 9 ];
```

```
 9
10    printf( "%s", "Enter a string: " );
11    scanf( "%c%8s", &x, y );
12
13    puts( "The input was:\n" );
14    printf( "the character \"%c\" and the string \"%s\"\n", x, y );
15 } // конец main
```

```
Enter a string: Sunday
The input was:
the character "S" and the string "unday"
```

Последовательность символов можно ввести с помощью **множества для сканирования**. Множество для сканирования – это множество символов, заключенных в квадратные скобки, [], которое следует за знаком процента в строке формата. Получив множество для сканирования, функция scanf сканирует входную строку в поисках символов, включенных во множество. Каждый такой символ сохраняется в аргументе, соответствующем множеству для сканирования, который должен быть указателем на массив символов. Сканирование прекращается, когда во входной строке встречается символ, не входящий во множество. Если первый символ во входном потоке *не* входит во множество для сканирования, аргумент-массив не изменяется. Программа в примере 9.15 использует множество [aeiou] для сканирования входного потока на наличие гласных букв. Обратите внимание, что инструкция читает первые семь букв из входного потока. Восьмая буква (h) отсутствует во множестве для сканирования, и поэтому сканирование завершается.

Пример 9.15 | Чтение с применением множества для сканирования

```
 1 // Пример 9.15: fig09_21.c
 2 // Чтение с применением множества для сканирования
 3 #include <stdio.h>
 4
 5 // выполнение программы начинается с функции main
 6 int main( void )
 7 {
 8    char z[ 9 ]; // определить массив z
 9
10    printf( "%s", "Enter string: " );
11    scanf( "%8[aeiou]", z ); // искать символы, входящие во множество
12
13    printf( "The input was \"%s\"\n", z );
14 } // конец main
```

```
Enter string: ooeeooahah
The input was "ooeeooa"
```

Множество для сканирования можно также использовать с целью поиска символов, отсутствующих во множестве, применив **инвертированное множество для сканирования**. Чтобы определить инвертированное множество для сканирования, добавьте после открывающей квадратной скобки

символ «крышки» (^). В результате в указанный массив будут сохраняться символы, отсутствующие во множестве для сканирования. Ввод завершается при встрече первого символа, присутствующего во множестве для сканирования. Программа в примере 9.16 использует инвертированное множество `[^aeiou]` для сканирования входного потока на наличие согласных букв.

Пример 9.16 | Чтение с применением инвертированного множества для сканирования

```
 1 // Пример 9.16: fig09_22.c
 2 // Чтение с применением инвертированного множества для сканирования
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    char z[ 9 ];
 8
 9    printf( "%s", "Enter a string: " );
10    scanf( "%8[^aeiou]", z ); // инвертированное множество
                                // для сканирования
11
12    printf( "The input was \"%s\"\n", z );
13 } // конец main
```

```
Enter a string: String
The input was "Str"
```

В спецификаторах преобразования можно указывать значение ширины поля, *чтобы ограничить ввод символов определенным их количеством*. Программа в примере 9.17 вводит последовательность цифр как двузначное целое число и остальные цифры как составляющие второе целое число.

Пример 9.17 | Чтение данных с указанием ширины поля

```
 1 // Пример 9.17: fig09_23.c
 2 // Чтение данных с указанием ширины поля
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    int x;
 8    int y;
 9
10    printf( "%s", "Enter a six digit integer: " );
11    scanf( "%2d%d", &x, &y );
12
13    printf( "The integers input were %d and %d\n", x, y );
14 } // конец main
```

```
Enter a six digit integer: 123456
The integers input were 12 and 3456
```

Часто бывает необходимо *пропустить* определенные символы во входном потоке. Например, пользователь может ввести дату так:

```
11-10-1999
```

Числа в такой дате нужно сохранить, а дефисы, разделяющие их, – пропустить. Чтобы пропустить ненужные символы, включите их в строку формата в вызове функции `scanf` (пробельные символы – такие как пробел, перевод строки или табуляция – позволяют пропускать любые пробельные символы). Например, чтобы пропустить дефисы в исходных данных, представленных выше, можно использовать такую инструкцию:

```
scanf( "%d-%d-%d", &month, &day, &year );
```

Эта инструкция прекрасно справится с дефисами, однако есть вероятность, что пользователь введет дату, как показано ниже:

```
10/11/1999
```

В этом случае предыдущая инструкция `scanf` *не* сможет пропустить ненужные символы. По этой причине в реализацию `scanf` была включена поддержка **символа подавления ввода** *. Этот символ позволяет функции `scanf` читать из входного потока данные любого типа и отбрасывать их, не присваивая какой-либо переменной. Программа в примере 9.18 использует символ подавления ввода в спецификаторе `%c`, чтобы показать, что символ в данной позиции во входном потоке следует прочитать и отбросить. Здесь сохраняются только номер дня, месяца и года. Для демонстрации корректности ввода программа выводит значения переменных, куда осуществлялся ввод. Списки аргументов в каждом вызове `scanf` не содержат переменных для спецификаторов, где используется символ подавления ввода. Соответствующие им символы просто отбрасываются.

Пример 9.18 | Чтение и пропуск символов во входном потоке

```
// Пример 9.18: fig09_24.c
// Чтение и пропуск символов во входном потоке
#include <stdio.h>

int main( void )
{
   int month1;
   int day1;
   int year1;
   int month2;
   int day2;
   int year2;

   printf( "%s", "Enter a date in the form mm-dd-yyyy: " );
   scanf( "%d%*c%d%*c%d", &month1, &day1, &year1 );

   printf( "month = %d day = %d year = %d\n\n", month1, day1, year1 );

   printf( "%s", "Enter a date in the form mm/dd/yyyy: " );
   scanf( "%d%*c%d%*c%d", &month2, &day2, &year2 );

```

```
22    printf( "month = %d day = %d year = %d\n", month2, day2, year2 );
23 } // конец main
```

```
Enter a date in the form mm-dd-yyyy: 11-18-2012
month = 11 day = 18 year = 2012

Enter a date in the form mm/dd/yyyy: 11/18/2012
month = 11 day = 18 year = 2012
```

9.12 Безопасное программирование на C

В стандарте языка C перечисляется множество ситуаций, когда неправильное использование аргументов в вызовах библиотечных функций может привести к неправильному поведению, что, в свою очередь, может способствовать появлению брешей в системе безопасности. Такие проблемы могут возникать при передаче функции `printf` (и любым другим ее разновидностям, таким как `sprintf`, `fprintf`, `printf_s` и др.) неправильно сформированных спецификаторов преобразования. Эти проблемы обсуждаются в рекомендации FIO00-C центра CERT (`www.securecoding.cert.org`), где также можно найти таблицу с допустимыми комбинациями флагов, модификаторов длины и символов спецификаторов, которые могут использоваться для формирования строки формата. В этой таблице приводятся типы аргументов, допустимые для каждого спецификатора. Вообще говоря, если в описании *любого* языка говорится, что некоторое действие может привести к неопределенному поведению, избегайте подобных действий, чтобы предотвратить проблемы с безопасностью.

10

Структуры, объединения, перечисления и поразрядные операции

В этой главе вы научитесь:

- создавать и использовать структуры, объединения и перечисления;
- передавать структуры функциям по значению и по ссылке;
- использовать объявление `typedef` для создания псевдонимов существующих имен типов;
- выполнять поразрядные операции с данными;
- создавать битовые поля, обеспечивающие более компактный способ хранения данных.

10.1 Введение

Структуры (иногда их также называют **агрегатами**) – это коллекции связанных между собой переменных, объединенных под одним общим именем. Структуры могут содержать переменные самых разных типов данных, в отличие от массивов, которые могут состоять только из элементов одинаковых типов. Структуры часто используются для определения *записей*, хранящихся в файлах (глава 11 «Файлы»). Указатели и структуры упрощают формирование еще более сложных структур данных, таких как связанные списки, очереди, стеки и деревья (глава12 «Структуры данных»). В этой главе мы также обсудим:

- объявление `typedef`, позволяющее создавать *псевдонимы* для типов данных, определенных ранее;
- объединения – типы данных, подобные структурам, в которых поля могут занимать одни и те же области памяти;
- поразрядные операторы, дающие возможность манипулировать отдельными разрядами (битами) целочисленных операндов;
- битовые поля – поля структур или объединений типа `unsigned int` или `int`, для которых определяется ширина в битах, помогающие обеспечить более компактное хранение информации;
- перечисления – множества целочисленных констант, представленных идентификаторами.

10.2 Определение структур

Структура – это **производный тип данных**, сконструированный из объектов других типов. Взгляните на следующее определение структуры:

```
struct card {
   char *face;
   char *suit;
}; // конец структуры card
```

Определение структуры начинается с ключевого слова `struct`. Идентификатор `card` – это имя (или тег) структуры, которое служит для идентификации структуры и используется вместе с ключевым словом `struct` в объявлениях переменных **структурного типа**, например `struct card`. Переменные, объявленные внутри фигурных скобок, называются **полями**, или **членами** структуры. Поля должны иметь уникальные имена в пределах одной структуры, но разные структуры могут содержать одноименные поля, при этом никаких конфликтов имен возникать не будет (причины этого описываются чуть ниже). Каждое определение структуры *должно* завершаться точкой с запятой.

Отсутствие точки с запятой в конце определения структуры считается синтаксической ошибкой.

Определение `struct card` содержит два поля `face` и `suit`, оба типа `char *`. Поля структур могут быть переменными любых простых типов (таких как `int`, `float` и др.) или составных (таких как массивы и другие структуры). Как было показано в главе 6, все элементы массива должны принадлежать к *одному* типу. Однако это правило не относится к полям структур. Например, следующая структура с информацией о работнике включает два поля с массивами символов, хранящими имя и фамилию, поле типа `unsigned int`, хранящее возраст, поле типа `char`, хранящее пол работника в виде символа `'M'` (Male – мужской) или `'F'` (Female – женский), и поле типа `double`, хранящее почасовую ставку оплаты:

```
struct employee {
   char firstName[ 20 ];
   char lastName[ 20 ];
   unsigned int age;
   char gender;
   double hourlySalary;
}; // конец структуры employee
```

10.2.1 Структуры со ссылками на самих себя

Структура не может хранить экземпляр самой себя. Например, переменная типа `struct employee` не может включать поле типа `struct employee`. Но переменные-указатели на тип `struct employee` вполне допустимы. Например:

```
struct employee2 {
   char firstName[ 20 ];
   char lastName[ 20 ];
   unsigned int age;
   char gender;
   double hourlySalary;
   struct employee2 person; // ОШИБКА
   struct employee2 *ePtr;  // указатель
}; // конец структуры employee2
```

Структура `struct employee2` содержит экземпляр самой себя (поле `person`), что представляет ошибку. Однако поле `ePtr` является указателем (типа `struct employee2 *`), и это вполне допустимо. Структуры, содержащие поле, являющееся указателем на структуру *того же* структурного типа, называют **структурами со ссылками на самих себя**. Такие структуры мы будем использовать в главе 12 для создания связанных списков.

10.2.2 Определение переменных структурных типов

Определения структур не резервируют место в памяти – каждое такое определение просто создает новый тип данных, который можно использовать в объявлениях переменных. Переменные структурных типов объявляются точно так же, как и переменные любых других типов. Например, объявление

```
struct card aCard, deck[ 52 ], *cardPtr;
```

создаст переменную `aCard` типа `struct card`, массив `deck` с 52 элементами типа `struct card` и указатель `cardPtr` на значение типа `struct card`. Переменные структурного типа могут также объявляться путем помещения их имен между закрывающей фигурной скобкой определения структуры и точкой с запятой. Например, предыдущее объявление переменных можно было бы объединить с определением структуры `struct card`, как показано ниже:

```
struct card {
   char *face;
   char *suit;
} aCard, deck[ 52 ], *cardPtr;
```

10.2.3 Имена структур

Имя структуры является необязательным. Если определение структуры не содержит ее имени, переменные данного структурного типа можно объявить *только* в определении структуры – объявить переменные этого структурного типа отдельно будет невозможно.

При создании новых структурных типов всегда давайте им имена. Имена структур могут пригодиться ниже в программе для создания новых переменных данного структурного типа.

10.2.4 Операции над структурами

Над структурами могут выполняться только следующие операции:

- присваивание структурных переменных переменным *того же* структурного типа;
- взятие адреса (&) переменной структурного типа;
- обращение к полям переменной структурного типа (раздел 10.4);
- применение оператора `sizeof` для определения размера переменной структурного типа.

Попытка присвоить переменной одного структурного типа значение переменной другого структурного типа вызовет ошибку компиляции.

Структуры нельзя сравнивать между собой с помощью операторов `==` и `!=`, потому что поля структур не всегда будут занимать непрерывную последовательность байтов в памяти. Иногда в структурах могут присутствовать «дырки» из-за того, что компьютеры могут хранить данные некоторых типов, только начиная с определенных адресов в памяти, например с границы машинного полуслова, слова или двойного слова. Обычно данные сохраняются в памяти по границам машинного слова – 2 или 4 байта. Взгляните на следующее определение структуры, где одновременно объявляются две переменные, `sample1` и `sample2`:

```
struct example {
   char c;
   int i;
} sample1, sample2;
```

В компьютере, где машинное слово имеет длину 2 байта, может потребоваться, чтобы каждое следующее поле структуры `struct example` начиналось с границы слова, то есть с четного адреса (размер слова зависит от аппаратной архитектуры компьютера). На рис. 10.1 показано (в побитовом представлении), как хранится в памяти переменная типа `struct example`, полям которой присвоены символ `'a'` и целое число `97`. Если поля структуры в памяти выравниваются по границам слов, в области памяти, где хранится переменная типа `struct example`, образуется ничем не занятая дырка размером 1 байт (байт с номером 1 на рис. 10.1). Значение байта дырки не определено. Даже если значения полей в переменных `sample1` и `sample2` фактически будут равны, структуры могут оказаться не равны из-за неопределенности значений в дырках.

Так как размеры элементов данных конкретного типа зависят от аппаратной архитектуры компьютера, а также из-за требований к выравниванию полей в памяти фактическое представление структур также зависит от аппаратной архитектуры.

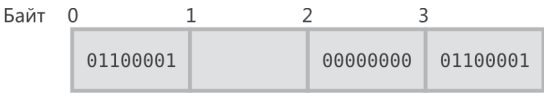


Рис. 10.1 | Из-за возможного выравнивания полей в переменной типа struct example может появиться «дырка»

10.3 Инициализация структур

Структуры могут инициализироваться с помощью списков инициализирующих значений, подобно массивам. Чтобы инициализировать переменную-структуру, вслед за ее именем в определении нужно добавить знак «равно» (=) и список инициализирующих значений, перечисленных через запятую, заключенный в фигурные скобки. Например, следующее объявление:

```
struct card aCard = { "Three", "Hearts" };
```

создаст переменную `aCard` типа `struct card` (объявленную в разделе 10.2) и инициализирует поле `face` значением `"Three"`, а поле `suit` – значением `"Hearts"`. Если инициализирующих значений в списке *меньше*, чем полей в структуре, остальные поля автоматически инициализируются нулевыми значениями (или значением `NULL`, если поле является указателем). Переменные-структуры, объявленные за пределами функций (то есть внешние переменные), инициализируются нулевыми значениями автоматически, если для них явно не указаны другие инициализаторы. Переменные-структуры могут также инициализироваться инструкциями присваивания, путем присваивания им значений переменных-структур *того же* типа или путем присваивания значений полям *по отдельности*.

10.4 Доступ к полям структур

Для доступа к полям структуры используются два оператора: оператор доступа к члену структуры (.) – известный также как **оператор «точки»** – и **оператор указателя в структуре** (->) – известный также как **оператор «стрелки»**. Оператор «точки» применяется для обращения к полю через имя переменной-структуры. Например, чтобы вывести значение поля `suit` переменной-структуры `aCard`, объявленной в разделе 10.3, можно использовать такую инструкцию:

```
printf( "%s", aCard.suit ); // выведет Hearts
```

Оператор указателя в структуре – состоящий из знака «минус» (-) и знака «больше» (>) без пробела между ними – обеспечивает доступ к полю

структуры через **указатель на эту структуру**. Допустим, что прежде был объявлен указатель `cardPtr` на значение типа `struct card` и ему был присвоен адрес переменной `aCard`. Чтобы вывести значение поля `suit` переменной `aCard` по указателю `cardPtr`, можно использовать такую инструкцию:

```
printf( "%s", cardPtr->suit ); // выведет Hearts
```

Выражение `cardPtr->suit` эквивалентно выражению `(*cardPtr).suit`, в котором сначала выполняется разыменование указателя, а затем осуществляется обращение к полю `suit` с помощью оператора «точка». Круглые скобки здесь необходимы, потому что оператор доступа к члену структуры (.) имеет более высокий приоритет, чем оператор разыменования указателя (*). Оператор указателя в структуре и оператор доступа к полю структуры, а также операторы круглых (осуществляющий вызов функций) и квадратных (`[]`) скобок (обеспечивающий доступ к элементам массивов по их индексам) имеют наивысший приоритет среди операторов и ассоциативность слева направо.

Не добавляйте пробелы вокруг операторов `->` и `.` — отсутствие пробелов поможет подчеркнуть, что выражения с этими операторами, по сути, являются именами переменных.

Добавление пробела между знаками `-` и `>`, образующими оператор «стрелки» (или между компонентами любого другого составного оператора, кроме `?:`), является синтаксической ошибкой.

Попытка сослаться на поле структуры, используя только имя поля, является синтаксической ошибкой.

Отказ от использования круглых скобок при попытке сослаться на поле структуры с помощью указателя и оператора «точки» (например, как `*cardPtr.suit`) является синтаксической ошибкой.

Программа в примере 10.1 демонстрирует применение операторов «точки» и «стрелки». С помощью оператора «точки» полям структуры `aCard` в этом примере присваиваются значения `"Ace"` и `"Spades"` соответственно (строки 18 и 19). Указателю `cardPtr` присваивается адрес структуры `aCard` (строка 21). Функция `printf` выводит значения полей структуры `aCard` с помощью оператора «точки» и имени переменной `aCard`, оператора «стрелки» и указателя `cardPtr`, а также оператора «точки» и выражения разыменования указателя `cardPtr` (строки с 23 по 25).

Пример 10.1 | Демонстрация применения операторов «точки» и «стрелки»

```
 1 // Пример 10.1: fig10_02.c
 2 // Демонстрация применения операторов
 3 // "точки" и "стрелки"
 4 #include <stdio.h>
 5
 6 // определение структуры card
 7 struct card {
 8    char *face; // определить указатель face
 9    char *suit; // определить указатель suit
10 }; // конец структуры card
11
12 int main( void )
13 {
14    struct card aCard;    // определить переменную типа struct card
15    struct card *cardPtr; // определить указатель на значение типа
                            // struct card
16
17    // сохранить строки в переменной aCard
18    aCard.face = "Ace";
19    aCard.suit = "Spades";
20
21    cardPtr = &aCard;    // присвоить адрес aCard указателю cardPtr
22
23    printf( "%s%s%s\n%s%s%s\n%s%s%s\n", aCard.face, " of ", aCard.suit,
24       cardPtr->face, " of ", cardPtr->suit,
25       ( *cardPtr ).face, " of ", ( *cardPtr ).suit );
26 } // конец main
```

```
Ace of Spades
Ace of Spades
Ace of Spades
```

10.5 Передача структур функциям

Структуры могут передаваться функциям в виде отдельных полей, целиком или по указателю на структуру. Когда функции передается структура целиком или отдельные ее поля, передача осуществляется *по значению*. То есть поля структуры в вызывающей функции не могут изменяться вызываемой функцией. Чтобы передать структуру *по ссылке*, следует передать адрес переменной-структуры. Массивы структур, подобно любым другим массивам, автоматически передаются по ссылке.

В главе 6 отмечалось, что массив можно передать по значению, используя структуру. Чтобы передать массив по значению, создайте структуру с полем-массивом. Структуры передаются по значению, поэтому и массивы в составе структур передаются по значению.

Предположение, что структуры, подобно массивам, автоматически передаются по ссылке, и попытки изменять структуры в вызываемых функциях являются логической ошибкой.

Передача структур по ссылке выполняется быстрее, чем передача по значению (требующая копирования структуры целиком).

10.6 typedef

Ключевое слово `typedef` представляет механизм создания синонимов (или псевдонимов) для прежде объявленных типов данных. Имена структурных типов часто определяются с помощью `typedef` с целью определить более короткие имена типов. Например, инструкция

```
typedef struct card Card;
```

определяет новый тип с именем `Card`, являющийся синонимом для типа `struct card`. Программисты на C часто используют `typedef` для определения структурных типов и потому объявляют безымянные структуры. Например, следующее определение:

```
typedef struct {
   char *face;
   char *suit;
} Card; // конец определения типа Card
```

объявляет структурный тип `Card` в одном объявлении со структурой.

Начинайте имена типов, определяемых с помощью `typedef`, с буквы верхнего регистра, чтобы подчеркнуть, что они являются синонимами для других имен типов.

Теперь имя типа `Card` можно использовать для объявления переменных типа `struct card`. Например, объявление:

```
Card deck[ 52 ];
```

определяет массив с 52 структурами `Card` (то есть с элементами типа `struct card`). Создание нового имени с помощью `typedef` не создает нового типа; `typedef` просто определяет новое имя для уже существующего типа, которое можно использовать как его псевдоним. Осмысленное имя поможет сделать программу самодокументируемой. Например, прочитав предыдущее объявление, мы сразу поймем, что «`deck` – это массив из 52 карт `Cards`».

Нередко ключевое слово `typedef` используется и для создания синонимов простых типов данных. Например, программа, требующая применения 4-байтовых целых чисел, может использовать тип `int` в одних системах и

тип `long` в других. Программы, проектируемые с учетом возможности переноса на разные платформы, могут использовать `typedef` для создания синонима 4-байтового типа, такого как `Integer`. Псевдоним `Integer` можно изменить лишь один раз в программе, чтобы обеспечить работоспособность программы в обеих системах.

Используйте `typedef`, чтобы сделать программу более переносимой.

Использование `typedef` может сделать программу более читаемой и упростить ее сопровождение.

10.7 Пример: высокопроизводительная программа перемешивания и раздачи колоды карт

Программа в примере 10.2 является продолжением примера тасования и раздачи колоды карт, обсуждавшегося в главе 7. Программа представляет колоду карт как массив структур и использует более высокопроизводительные алгоритмы тасования и раздачи. Вывод программы представлен в примере 10.3.

Пример 10.2 | Программа тасования и раздачи карт, основанная на применении структур

```
// Пример 10.2: fig10_03.c
// Программа тасования и раздачи карт, основанная на применении структур
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define CARDS 52
#define FACES 13

// определение структуры card
struct card {
   const char *face; // определить указатель face
   const char *suit; // определить указатель suit
}; // конец структуры card

typedef struct card Card; // новое имя типа для struct card

// прототипы
void fillDeck( Card * const wDeck, const char * wFace[],
   const char * wSuit[] );
void shuffle( Card * const wDeck );
void deal( const Card * const wDeck );

```

```
24 int main( void )
25 {
26    Card deck[ CARDS ]; // определить массив элементов типа Cards
27
28    // инициализировать массив указателей
29    const char *face[] = { "Ace", "Deuce", "Three", "Four", "Five",
30                           "Six", "Seven", "Eight", "Nine", "Ten",
31                           "Jack", "Queen", "King"};
32
33    // инициализировать массив указателей
34    const char *suit[] = { "Hearts", "Diamonds", "Clubs", "Spades"};
35
36    srand( time( NULL ) ); // переустановить генератор случайных чисел
37
38    fillDeck( deck, face, suit ); // заполнить колоду значениями типа Cards
39    shuffle( deck );              // перетасовать колоду карт Cards
40    deal( deck );                 // раздать все 52 карты Cards
41 } // конец main
42
43 // помещает строки в структуры Card
44 void fillDeck( Card * const wDeck, const char * wFace[],
45    const char * wSuit[] )
46 {
47    size_t i; // счетчик
48
49    // цикл по элементам массива wDeck
50    for ( i = 0; i < CARDS; ++i ) {
51       wDeck[ i ].face = wFace[ i % FACES ];
52       wDeck[ i ].suit = wSuit[ i / FACES ];
53    } // конец for
54 } // конец функции fillDeck
55
56 // тасует карты
57 void shuffle( Card * const wDeck )
58 {
59    size_t i;  // счетчик
60    size_t j;  // переменная для хранения случайного числа 0-51
61    Card temp; // временная структура для обмена карт местами
62
63    // цикл по элементам массива wDeck и случайный обмен местами
64    for ( i = 0; i < CARDS; ++i ) {
65       j = rand() % CARDS;
66       temp = wDeck[ i ];
67       wDeck[ i ] = wDeck[ j ];
68       wDeck[ j ] = temp;
69    } // конец for
70 } // конец функции shuffle
71
72 // раздает карты
73 void deal( const Card * const wDeck )
74 {
75    size_t i; // счетчик
76
77    // цикл по элементам массива wDeck
78    for ( i = 0; i < CARDS; ++i ) {
79       printf( "%5s of %-8s%s", wDeck[ i ].face, wDeck[ i ].suit,
80          ( i + 1 ) % 4 ? " " : "\n" );
81    } // конец for
82 } // конец функции deal
```

Пример 10.3 | Вывод программы тасования и раздачи карт

```
Three of Hearts     Jack of Clubs     Three of Spades      Six of
Diamonds
 Five of Hearts    Eight of Spades    Three of Clubs     Deuce of Spades
 Jack of Spades     Four of Hearts    Deuce of Hearts      Six of Clubs
Queen of Clubs     Three of Diamonds  Eight of Diamonds   King of Clubs
 King of Hearts    Eight of Hearts    Queen of Hearts    Seven of Clubs
Seven of Diamonds   Nine of Spades     Five of Clubs     Eight of Clubs
  Six of Hearts    Deuce of Diamonds   Five of Spades     Four of Clubs
Deuce of Clubs      Nine of Hearts    Seven of Hearts     Four of Spades
  Ten of Spades     King of Diamonds    Ten of Hearts     Jack of
Diamonds
 Four of Diamonds    Six of Spades     Five of Diamonds    Ace of
Diamonds
  Ace of Clubs      Jack of Hearts      Ten of Clubs     Queen of
Diamonds
  Ace of Hearts      Ten of Diamonds   Nine of Clubs      King of Spades
  Ace of Spades     Nine of Diamonds  Seven of Spades    Queen of Spades
```

Функция `fillDeck` (строки 44–54) в программе инициализирует массив элементов типа `Card` в порядке от `"Ace"` (туза) до `"King"` (короля) каждой масти. Затем массив передается (строка 39) функции `shuffle` (строки 57–70), реализующей высокопроизводительный алгоритм тасования карт. Функция `shuffle` принимает массив с 52 картами типа `Cards`, выполняет обход элементов массива в цикле (строки 64–69) и для каждой карты выбирает случайное число в диапазоне от 0 до 51. Затем текущий и случайно выбранный элементы массива меняются местами (строки 66–68). Всего за один проход выполняются 52 перемены мест, и массив карт оказывается перетасован! Этот алгоритм *не* страдает проблемой *переноса завершения на неопределенное время* (*indefinite postponement*), как представленный в главе 7. Так как элементы массива меняются местами непосредственно в массиве, высокопроизводительному алгоритму раздачи, реализованному в функции `deal` (строки 73–82), достаточно всего одного прохода через массив, чтобы раздать перетасованные карты.

Если при ссылке на отдельную структуру в массиве структур забыть добавить индекс, это приведет к появлению синтаксической ошибки.

10.8 Объединения

Объединение (union) – это *производный тип данных*, подобный структурам, поля которого *занимают одну и ту же область памяти*. В одних ситуациях лучше могут подходить одни переменные, в других – другие. Поэтому в свое время были изобретены объединения, позволяющие экономно использовать одну и ту же область памяти для разных переменных в разных ситуациях. Пол объединения может иметь *любой* тип данных. Количество байтов, исполь-

зуемое для хранения экземпляра объединения, равно величине *наибольшего* поля. В большинстве случаев объединения содержат два или более типов данных. В каждый конкретный момент обратиться можно только к одному типу данных. Обеспечение корректности использования типов в объединениях полностью возлагается на программиста.

Обращение к данным в объединении с помощью переменной неверного типа является логической ошибкой.

Результат операции извлечения данных из объединения с другим типом, отличным от того, с каким они сохранялись, зависит от реализации.

10.8.1 Объявление объединений

Определение объединения следует тому же формату, что и определение структуры. Следующее определение объединения:

```
union number {
   int x;
   double y;
}; // конец объединения number
```

сообщает, что `number` – это объединение полей `int x` и `double y`. Обычно определения объединений помещаются в заголовочные файлы, которые подключаются в файлах с исходными текстами, где используются эти объединения.

Как и определения структур, определения объединений просто создают новые типы. Размещение определения объединения или структуры за пределами функций не приводит к созданию глобальных переменных.

10.8.2 Операции над объединениями

Над объединениями могут выполняться следующие операции: присваивание одного объединения другому объединению того же типа, взятие адреса (&) переменной-объединения и обращение к полям объединения с помощью операторов «точки» и «стрелки». Объединения не могут сравниваться с помощью операторов == и != по тем же причинам, что и структуры.

10.8.3 Инициализация объединений в объявлениях

Для инициализации экземпляра объединения *можно использовать значение того же типа, что и первое поле объединения.* Например, для объединения из раздела 10.8.1 инструкция

```
union number value = { 10 };
```

представляет допустимый способ инициализации переменной-объединения, потому что она инициализируется значением типа `int`. Но следующая инструкция выполнит усечение дробной части вещественного числа и вызовет появление предупреждения компилятора:

```
union number value = { 1.43 };
```

Объем памяти, необходимый для хранения объединения, зависит от реализации, но он никогда не будет меньше наибольшего поля объединения.

Некоторые объединения не переносятся непосредственно в другие системы. Переносимость объединения часто зависит от правил выравнивания, применяемых к типам данных, составляющих объединение, в этой системе.

10.8.4 Демонстрация объединений

Программа в примере 10.4 использует переменную `value` (строка 13) типа `union number` (строки 6–9) для отображения значения, хранящегося в объединении, как значения типа `int` и `double`. Вывод программы *зависит от реализации*. Из полученного вывода видно, что внутреннее представление значения типа `double` может существенно отличаться от представления значения типа `int`.

Пример 10.4 | Вывод значений обоих полей одного объединения

```
 1 // Пример 10.4: fig10_05.c
 2 // Вывод значений обоих полей одного объединения
 3 #include <stdio.h>
 4
 5 // определение объединения number
 6 union number {
 7    int x;
 8    double y;
 9 }; // конец объединения number
10
11 int main( void )
12 {
13    union number value; // определить переменную-объединение
14
15    value.x = 100;      // сохранить целое число в объединении
16    printf( "%s\n%s\n%s\n %d\n\n%s\n %f\n\n\n",
17       "Put 100 in the integer member",
18       "and print both members.",
19       "int:", value.x,
20       "double:", value.y );
21
22    value.y = 100.0; // сохранить вещественное число в том же объединении
23    printf( "%s\n%s\n%s\n %d\n\n%s\n %f\n",
24       "Put 100.0 in the floating member",
25       "and print both members.",
26       "int:", value.x,
```

```
27       "double:", value.y );
28 } // конец main
```

```
Put 100 in the integer member
and print both members.
int:
  100

double:
  -92559592117433136000000000000000000000000000000000000000000000000.0000
00

Put 100.0 in the floating member
and print both members.
int:
  0

double:
  100.000000
```

10.9 Поразрядные операторы

Компьютеры хранят данные как последовательности битов (или разрядов). Каждый бит может принимать два значения, `0` или `1`. В большинстве систем последовательность из 8 бит образует байт – единицу хранения информации для переменной типа `char`. Данные других типов занимают в памяти большее количество байтов. Поразрядные операторы используются для выполнения операций с отдельными разрядами, или битами, операндов целочисленных типов как со знаком, так и без знака, но, как правило, поразрядные операторы применяются к целочисленным значениям без знака.

Поразрядные операции с данными тесно связаны с аппаратной архитектурой компьютера.

Поразрядные операторы, обсуждаемые в этом разделе, оперируют с двоичным представлением целочисленных операндов. Более подробное описание двоичного представления чисел (то есть представления в системе счисления с основанием 2) можно найти в приложении С. Из-за аппаратнозависимой природы поразрядных операций программы, использующие их, в некоторых системах могут производить неправильные результаты.

К поразрядным операторам относятся: **поразрядное И** (&), **поразрядное ИЛИ** (|), **поразрядное исключающее ИЛИ** (^), **сдвиг влево** (<<), **сдвиг вправо** (>>) и **дополнение** (~). Поразрядные операторы И, ИЛИ и исключающее ИЛИ сравнивают свои операнды бит за битом. Оператор *поразрядного И* устанавливает бит в результате в значение 1, только если соответствующие биты в обоих операндах имеют значение 1. Оператор *поразрядного ИЛИ* устанавливает бит в результате в значение 1, если хотя бы в одном из

операндов соответствующий бит имеет значение 1. Оператор *поразрядного исключающего ИЛИ* устанавливает бит в результате в значение 1, только если в одном операнде соответствующий бит имеет значение 1, а в другом этот же бит имеет значение 0. Оператор *поразрядного сдвига влево* сдвигает влево биты левого операнда на количество позиций, определяемое правым операндом. Оператор *поразрядного сдвига вправо* сдвигает вправо биты левого операнда на количество позиций, определяемое правым операндом. Оператор *поразрядного дополнения* инвертирует значения битов в своем операнде, меняя 0 на 1, а 1 на 0. Подробное обсуждение всех поразрядных операторов приводится в примерах ниже, а в табл. 10.1 дается краткое их описание.

Таблица 10.1 | Поразрядные операторы

Оператор	Описание
& поразрядное И	Бит в результате устанавливается в значение 1, только если соответствующие биты в обоих операндах имеют значение 1
| поразрядное ИЛИ	Бит в результате устанавливается в значение 1, если хотя бы в одном из операндов соответствующий бит имеет значение 1
^ поразрядное исключающее ИЛИ	Бит в результате устанавливается в значение 1, только если в одном операнде соответствующий бит имеет значение 1, а в другом этот же бит имеет значение 0
<< сдвиг влево	Сдвигает влево биты первого операнда на количество позиций, определяемое вторым операндом. Заполняет биты справа значением 0
>> сдвиг вправо	Сдвигает вправо биты первого операнда на количество позиций, определяемое вторым операндом. Порядок заполнения битов слева зависит от аппаратной архитектуры, когда первый операнд представлен отрицательным значением
~ поразрядное дополнение	Инвертирует значения битов операнда, меняя 0 на 1, а 1 на 0

10.9.1 Вывод целых чисел без знака в двоичном представлении

При использовании поразрядных операторов бывает полезно выводить результаты в двоичном представлении, чтобы видеть точный эффект применения операторов. Программа в примере 10.5 выводит значение типа `unsigned int` в двоичном представлении, группами по восемь бит для удобочитаемости. Во всех примерах, что приводятся далее в этой главе, предполагается, что значения типа `int` занимают в памяти 4 байта (32 бита).

Пример 10.5 | Вывод значения типа unsigned int в двоичном представлении

```
1 // Пример 10.5: fig10_07.c
2 // Вывод значения типа unsigned int в двоичном представлении
3 #include <stdio.h>
```

```
 4
 5 void displayBits( unsigned int value ); // прототип
 6
 7 int main( void )
 8 {
 9    unsigned int x; // переменная для хранения ввода пользователя
10
11    printf( "%s", "Enter a nonnegative int: " );
12    scanf( "%u", &x );
13
14    displayBits( x );
15 } // конец main
16
17 // выводит значение типа unsigned int в двоичном представлении
18 void displayBits( unsigned int value )
19 {
20    unsigned int c; // счетчик
21
22    // определить значение displayMask как 1, сдвинутое влево на 31 позицию
23    unsigned int displayMask = 1 << 31;
24
25    printf( "%10u = ", value );
26
27    // выполнить обход всех битов
28    for ( c = 1; c <= 32; ++c ) {
29       putchar( value & displayMask ? '1' : '0' );
30       value <<= 1; // сдвинуть влево на 1 позицию
31
32       if ( c % 8 == 0 ) { // вывести пробел после каждого 8 бита
33          putchar( ' ' );
34       } // конец if
35    } // конец for
36
37    putchar( '\n' );
38 } // конец функции displayBits
```

```
Enter a nonnegative int: 65000
     65000 = 00000000 00000000 11111101 11101000
```

Функция `displayBits` (строки 18–38) использует оператор поразрядного И для объединения значения переменной `value` со значением переменной `displayMask` (строка 29). Часто оператор поразрядного И используется с операндом, который называют **маской** – целочисленным значением с определенными битами, установленными в значение 1. Маски используются для *сокрытия* одних и *проверки* других битов в проверяемом значении. Переменной `displayMask` в функции `displayBits` присваивается значение

```
1 << 31    (10000000 00000000 00000000 00000000)
```

Оператор `<<` сдвигает значение 1 из самого младшего бита (самого правого) в самый старший бит (самый левый) и заполняет биты справа значением 0. Строка 29

```
29       putchar( value & displayMask ? '1' : '0' );
```

определяет, какой символ должен выводиться, `'1'` или `'0'`, для текущего левого бита в значении переменной. Когда оператор `&` применяет операнд `displayMask` к значению `value`, все биты, кроме самого старшего, оказываются «замаскированными», потому что операция «И» любого бита со значением 0 дает в результате 0. Если самый левый бит оказывается равен `1`, выражение `value & displayMask` возвращает ненулевое (истинное) значение и на экран выводится `1`, в противном случае выводится `0`. После этого значение переменной `value` сдвигается влево на одну позицию выражением `value <<= 1` (эквивалентно выражению `value = value << 1`). Эти шаги повторяются для каждого бита в переменной `value`. В табл. 10.2 приводятся результаты выполнения операции поразрядного И с любыми возможными комбинациями значений двух битов.

Использование операторов логического ИЛИ (`&&`) вместо поразрядного ИЛИ (`&`) является логической ошибкой.

Таблица 10.2 | Результаты операции поразрядного И с любыми возможными комбинациями значений двух битов

Бит 1	Бит 2	Бит 1 & Бит 2
0	0	0
0	1	0
1	0	0
1	1	1

10.9.2 Повышение переносимости и масштабируемости функции displayBits

В строке 23 в примере 10.5 жестко задано значение `31`, определяющее количество позиций, на которое нужно сдвинуть влево значение `1`, чтобы установить самый старший бит в переменной `displayMask`. Аналогично в строке 28 жестко задано значение `32`, определяющее количество итераций цикла – по одной на каждый бит в переменной `value`. В данной программе предполагается, что значение типа `unsigned int` всегда занимает в памяти 32 бита (4 байта). Однако многие современные компьютеры построены на аппаратной архитектуре, в которой одно машинное слово имеет размер 64 бита. Как и любой программист на C, вы будете стремиться обеспечить совместимость своих программ со всеми аппаратными архитектурами, в которых значения типа `unsigned int` могут занимать меньшее или большее количество битов.

Мы можем сделать программу в примере 10.5 более масштабируемой и переносимой, заменив целое число `31` в строке 23 выражением

```
CHAR_BIT * sizeof( unsigned int ) - 1
```

и число 32 в строке 28 выражением

```
CHAR_BIT * sizeof( unsigned int )
```

Символическая константа `CHAR_BIT` (определена в заголовочном файле `<limits.h>`) представляет количество битов в одном байте (обычно 8). Как уже говорилось в разделе 7.7, оператор `sizeof` определяет количество байтов, необходимое для хранения указанного объекта или значения указанного типа. В компьютере, где машинное слово занимает 32 бита, выражение `sizeof( unsigned int )` вернет значение `4`, поэтому два выражения выше вернут значения `31` и `32` соответственно. В компьютере, где машинное слово занимает 16 бит, выражение `sizeof( unsigned int )` вернет значение `2`, и два выражения выше вернут значения `15` и `16` соответственно.

10.9.3 Поразрядные операторы «И», «ИЛИ», исключающее «ИЛИ» и дополнение

Программа в примере 10.6 демонстрирует использование поразрядных операторов И, ИЛИ, исключающего ИЛИ и дополнения. Для вывода результатов программа использует функцию `displayBits` (строки 51–71). Вывод программы представлен в примере 10.7.

Пример 10.6 | Демонстрация поразрядных операторов И, ИЛИ, исключающее ИЛИ и дополнение

```
 1 // Пример 10.6: fig10_09.c
 2 // Демонстрация поразрядных операторов И, ИЛИ,
 3 // исключающее ИЛИ и дополнение
 4 #include <stdio.h>
 5
 6 void displayBits( unsigned int value ); // прототип
 7
 8 int main( void )
 9 {
10    unsigned int number1;
11    unsigned int number2;
12    unsigned int mask;
13    unsigned int setBits;
14
15    // демонстрация оператора поразрядного И (&)
16    number1 = 65535;
17    mask = 1;
18    puts( "The result of combining the following" );
19    displayBits( number1 );
20    displayBits( mask );
21    puts( "using the bitwise AND operator & is" );
22    displayBits( number1 & mask );
23
24    // демонстрация оператора поразрядного ИЛИ (|)
25    number1 = 15;
```

```
26    setBits = 241;
27    puts( "\nThe result of combining the following" );
28    displayBits( number1 );
29    displayBits( setBits );
30    puts( "using the bitwise inclusive OR operator | is" );
31    displayBits( number1 | setBits );
32
33    // демонстрация оператора поразрядного исключающего ИЛИ (^)
34    number1 = 139;
35    number2 = 199;
36    puts( "\nThe result of combining the following" );
37    displayBits( number1 );
38    displayBits( number2 );
39    puts( "using the bitwise exclusive OR operator ^ is" );
40    displayBits( number1 ^ number2 );
41
42    // демонстрация оператора поразрядного дополнения (~)
43    number1 = 21845;
44    puts( "\nThe one's complement of" );
45    displayBits( number1 );
46    puts( "is" );
47    displayBits( ~number1 );
48 } // конец main
49
50 // выводит значение типа unsigned int в двоичном представлении
51 void displayBits( unsigned int value )
52 {
53    unsigned int c; // счетчик
54
55    // объявить displayMask и сдвинуть влево на 31 разряд
56    unsigned int displayMask = 1 << 31;
57
58    printf( "%10u = ", value );
59
60    // выполнить обход всех битов
61    for ( c = 1; c <= 32; ++c ) {
62       putchar( value & displayMask ? '1' : '0' );
63       value <<= 1; // сдвинуть влево на 1 позицию
64
65       if ( c % 8 == 0 ) { // вывести пробел после каждого 8 бита
66          putchar( ' ' );
67       } // конец if
68    } // конец for
69
70    putchar( '\n' );
71 } // конец функции displayBits
```

Пример 10.7 | Вывод программы из примера 10.6

```
The result of combining the following
     65535 = 00000000 00000000 11111111 11111111
         1 = 00000000 00000000 00000000 00000001
using the bitwise AND operator & is
         1 = 00000000 00000000 00000000 00000001
```

```
The result of combining the following
        15 = 00000000 00000000 00000000 00001111
       241 = 00000000 00000000 00000000 11110001
using the bitwise inclusive OR operator | is
       255 = 00000000 00000000 00000000 11111111

The result of combining the following
       139 = 00000000 00000000 00000000 10001011
       199 = 00000000 00000000 00000000 11000111
using the bitwise exclusive OR operator ^ is
        76 = 00000000 00000000 00000000 01001100

The one's complement of
     21845 = 00000000 00000000 01010101 01010101
is
4294945450 = 11111111 11111111 10101010 10101010
```

Сначала переменной `number1` в программе из примера 10.6 присваивается значение `65535` (`00000000 00000000 11111111 11111111`), а переменной `mask` – значение `1` (`00000000 00000000 00000000 00000001`). Затем со значениями `number1` и `mask` выполняется *операция поразрядного И* (`&`) в выражении `number1 & mask` (строка 22), в результате получается значение `00000000 00000000 00000000 00000001`. Все биты, кроме самого младшего, в переменной `number1` оказались замаскированы значением переменной `mask`.

Оператор поразрядного ИЛИ используется для установки определенных битов в значение 1. В примере 10.6 в строке 25 переменной `number1` присваивается значение `15` (`00000000 00000000 00000000 00001111`), а переменной `setBits` – значение `241` (`00000000 00000000 00000000 11110001`) в строке 26. Когда к переменным `number1` и `setBits` применяется *оператор поразрядного ИЛИ*, в выражении `number1 | setBits` (строка 31) в результате получается значение `255` (`00000000 00000000 00000000 11111111`). В табл. 10.3 приводятся результаты выполнения операции поразрядного ИЛИ с любыми возможными комбинациями значений двух битов.

Таблица 10.3 | Результаты операции поразрядного ИЛИ с любыми возможными комбинациями значений двух битов

Бит 1	Бит 2	Бит 1 | Бит 2
0	0	0
0	1	1
1	0	1
1	1	1

Оператор поразрядного исключающего ИЛИ (`^`) устанавливает биты в значение 1 в результате, только если в одном из операндов соответствующий бит имеет значение 1, а в другом этот же бит имеет значение 0. В примере 10.6 переменным `number1` и `number2` присваиваются значения

139 (00000000 00000000 00000000 10001011) и 199 (00000000 00000000 00000000 11000111) в строках 34–35. Когда к ним применяется *оператор поразрядного исключающего ИЛИ*, в выражении number1 ^ number2 (строка 40) в результате получается значение 00000000 00000000 00000000 01001100. В табл. 10.4 приводятся результаты выполнения операции поразрядного исключающего ИЛИ с любыми возможными комбинациями значений двух битов.

Таблица 10.3 | Результаты операции поразрядного исключающего ИЛИ с любыми возможными комбинациями значений двух битов

Бит 1	Бит 2	Бит 1 ^ Бит 2
0	0	0
0	1	1
1	0	1
1	1	0

Оператор **поразрядного дополнения** (~) инвертирует значения битов в своем операнде, меняя 0 на 1 и 1 на 0, – эта операция также называется «дополнением до единицы». В примере 10.6 переменной number1 присваивается значение 21845 (00000000 00000000 01010101 01010101) в строке 43. Когда к ней применяется оператор поразрядного дополнения, в выражении ~number1 (строка 47) в результате получается значение 11111111 11111111 10101010 10101010.

10.9.4 Использование операторов поразрядного сдвига влево и вправо

Программа в примере 10.8 демонстрирует применение операторов *сдвига влево* (<<) и *сдвига вправо* (>>). Для вывода целочисленных значений в этой программе используется функция displayBits.

Пример 10.8 | Демонстрация операторов поразрядного сдвига

```
1 // Пример 10.8: fig10_13.c
2 // Демонстрация операторов поразрядного сдвига
3 #include <stdio.h>
4
5 void displayBits( unsigned int value ); // прототип
6
7 int main( void )
8 {
9    unsigned int number1 = 960; // инициализировать number1
10
11    // продемонстрировать сдвиг влево
12    puts( "\nThe result of left shifting" );
13    displayBits( number1 );
14    puts( "8 bit positions using the left shift operator << is" );
```

```
15    displayBits( number1 << 8 );
16
17    // продемонстрировать сдвиг вправо
18    puts( "\nThe result of right shifting" );
19    displayBits( number1 );
20    puts( "8 bit positions using the right shift operator >> is" );
21    displayBits( number1 >> 8 );
22 } // конец main
23
24 // выводит значение типа unsigned int в двоичном представлении
25 void displayBits( unsigned int value )
26 {
27    unsigned int c; // счетчик
28
29    // объявить displayMask и сдвинуть влево на 31 разряд
30    unsigned int displayMask = 1 << 31;
31
32    printf( "%7u = ", value );
33
34    // выполнить обход всех битов
35    for ( c = 1; c <= 32; ++c ) {
36       putchar( value & displayMask ? '1' : '0' );
37       value <<= 1; // сдвинуть значение value влево на 1 разряд
38
39       if ( c % 8 == 0 ) { // вывести пробел после каждого 8 бита
40          putchar( ' ' );
41       } // конец if
42    } // конец for
43
44    putchar( '\n' );
45 } // конец функции displayBits
```

```
The result of left shifting
    960 = 00000000 00000000 00000011 11000000
8 bit positions using the left shift operator << is
 245760 = 00000000 00000011 11000000 00000000

The result of right shifting
    960 = 00000000 00000000 00000011 11000000
8 bit positions using the right shift operator >> is
      3 = 00000000 00000000 00000000 00000011
```

Оператор поразрядного сдвига влево (`<<`) сдвигает влево биты левого операнда на количество позиций, определяемое правым операндом. Освободившиеся биты справа заполняются нулями; единицы, сдвинутые влево за границы исходного значения, теряются безвозвратно. В строке 9 в примере 10.8 переменной `number1` присваивается значение `960` (`00000000 00000000 00000011 11000000`). В результате сдвига переменной `number1` влево на 8 позиций в выражении `number1 << 8` (строка 15) получается значение `49152` (`00000000 00000011 11000000 00000000`).

Оператор поразрядного сдвига вправо (`>>`) сдвигает вправо биты левого операнда на количество позиций, определяемое правым операндом. Если выполняется сдвиг значения `unsigned int`, освободившиеся биты слева заполня-

ются нулями; единицы, сдвинутые вправо за границы исходного значения, теряются безвозвратно. В результате сдвига переменной `number1` вправо на 8 позиций в выражении `number1 >> 8` (строка 21) получается значение `3` (`00000000 00000000 00000000 00000011`).

Результат операции сдвига влево или вправо не определен, если правый операнд имеет отрицательное значение или его значение превышает количество битов в левом операнде.

Результат операции сдвига отрицательного значения вправо зависит от реализации.

10.9.5 Операторы поразрядного присваивания

Для каждого поразрядного оператора имеется соответствующий оператор присваивания. Все **операторы поразрядного присваивания** перечислены в табл. 10.5 и используются аналогично подобным арифметическим операторам присваивания, представленным в главе 3.

Таблица 10.5 | Операторы поразрядного присваивания

Оператор	Описание
`&=`	Присваивание с поразрядной операцией И
`|=`	Присваивание с поразрядной операцией ИЛИ
`^=`	Присваивание с поразрядной операцией исключающее ИЛИ
`<<=`	Присваивание с поразрядным сдвигом влево
`>>=`	Присваивание с поразрядным сдвигом влево

В табл. 10.6 указаны приоритеты и ассоциативность операторов, с которыми мы познакомились к данному моменту. Операторы перечислены сверху вниз, в порядке уменьшения приоритетов.

Таблица 10.6 | Приоритеты и ассоциативность операторов

Операторы	Ассоциативность	Тип
`[] () . -> ++` (*постфиксный*) `--` (*постфиксный*)	Справа налево	Постфиксный
`+ - ++ -- ! * & ~ sizeof` (*тип*)	Справа налево	Унарный
`* / %`	Слева направо	Мультипликативный
`+ -`	Слева направо	Аддитивный
`<< >>`	Слева направо	Сдвиговый
`< <= > >=`	Слева направо	Проверка отношения
`== !=`	Слева направо	Проверка на равенство
`&`	Слева направо	Поразрядное И

Таблица 10.6 | (*окончание*)

Операторы	Ассоциативность	Тип
^	Слева направо	Поразрядное исключающее ИЛИ
|	Слева направо	Поразрядное ИЛИ
&&	Слева направо	Логическое И
||	Слева направо	Логическое ИЛИ
?:	Справа налево	Условный
= += -= *= /= %= &= |= ^= <<= >>=	Справа налево	Присваивание
,	Слева направо	Запятая

10.10 Битовые поля

Язык C позволяет указывать количество битов, занимаемых полями типа `unsigned int` или `int` в структуре или объединении. Такие поля называются **битовыми полями**. Битовые поля обеспечивают более экономное использование памяти для хранения данных. Битовые поля должны объявляться как значения типа `int` или `unsigned int`.

Битовые поля позволяют экономить память.

Взгляните на следующее определение структуры:

```
struct bitCard {
   unsigned int face  : 4;
   unsigned int suit  : 2;
   unsigned int color : 1;
}; // конец структуры bitCard
```

содержащей три битовых поля типа `unsigned int` – `face`, `suit` и `color` – используемые для представления колоды из 52 карт. Битовое поле определяется двоеточием (:), следующим за **именем поля**, и целочисленной константой, определяющей ширину поля в битах. Значение ширины должно быть целым числом в диапазоне от 0 до общего количества битов в значениях типа `int` в данной системе включительно. Все наши примеры мы тестировали в системе, где целые числа занимают в памяти 4 байта (32 бита).

Предыдущее определение структуры указывает, что поле `face` занимает 4 бита, поле `suit` занимает 2 бита и поле `color` занимает 1 бит. Количество битов было определено из диапазона возможных значений для каждого поля в структуре. Поле `face` должно хранить значения в диапазоне от 0 (Ace – Туз) до 12 (King – Король) – 4 битами можно представить значения в диапазоне 0–15. Поле `suit` должно хранить значения от 0 до 3 (0 = Diamonds (бубны), 1 = Hearts (черви), 2 = Clubs (трефы), 3 = Spades

(пики)) – 2 битами можно представить значения в диапазоне 0–3. Наконец, поле `color` должно хранить всего два значения – 0 (красный) или 1 (черный) – 1 битом можно представить два значения, 0 и 1.

Программа в примере 10.9 (вывод этой программы приводится в примере 10.10) создает в строке 20 массив `deck`, содержащий 52 структуры `struct bitCard`. Функция `fillDeck` (строки 27–37) заполняет массив `deck` информацией о 52 картах, а функция `deal` (строк 41–53) выводит содержимое этого массива. Обратите внимание, что доступ к битовым полям структуры выполняется точно так же, как доступ к любым другим полям структур. Поле `color` было добавлено с целью отображения цвета масти карты в системах, снабженных цветным дисплеем. Существует возможность определять **неименованные битовые поля**, играющие роль **дополнений** в структуре. Например, в следующем определении структуры:

```
struct example {
   unsigned int a : 13;
   unsigned int   : 19;
   unsigned int b : 4;
}; // конец структуры example
```

используется неименованное битовое поле шириной 19 бит, играющее роль дополнения – в нем ничего не хранится. Поле `b` (в системах с 4-байтным машинным словом) будет храниться в отдельном машинном слове.

Пример 10.9 | Представление игральных карт с помощью битовых полей в структуре

```
 1 // Пример 10.9: fig10_16.c
 2 // Представление игральных карт с помощью битовых полей в структуре
 3 #include <stdio.h>
 4 #define CARDS 52
 5
 6 // определение структуры bitCard с битовыми полями
 7 struct bitCard {
 8    unsigned int face  : 4; // 4 бита; 0-15
 9    unsigned int suit  : 2; // 2 бита; 0-3
10    unsigned int color : 1; // 1 бит; 0-1
11 }; // конец структуры bitCard
12
13 typedef struct bitCard Card; // новое имя типа для struct bitCard
14
15 void fillDeck( Card * const wDeck );   // прототип
16 void deal( const Card * const wDeck ); // прототип
17
18 int main( void )
19 {
20    Card deck[ CARDS ]; // создать массив структур Cards
21
22    fillDeck( deck );
23    deal( deck );
24 } // конец main
25
26 // инициализирует массив Cards
```

```
27 void fillDeck( Card * const wDeck )
28 {
29    size_t i; // счетчик
30
31    // цикл по элементам массива wDeck
32    for ( i = 0; i < CARDS; ++i ) {
33       wDeck[ i ].face  = i % (CARDS / 4);
34       wDeck[ i ].suit  = i / (CARDS / 4);
35       wDeck[ i ].color = i / (CARDS / 4);
36    } // конец for
37 } // конец функции fillDeck
38
39 // выводит карты в две колонки; карты с индексами 0-25 — в колонку 1
40 // карты с индексами 26-51 — в колонку 2
41 void deal( const Card * const wDeck )
42 {
43    size_t k1; // индексы 0-25
44    size_t k2; // индексы 26-51
45
46    // цикл по элементам массива wDeck
47    for ( k1 = 0, k2 = k1 + 26; k1 < CARDS / 2; ++k1, ++k2 ) {
48       printf( "Card:%3d Suit:%2d Color:%2d ",
49          wDeck[ k1 ].face, wDeck[ k1 ].suit, wDeck[ k1 ].color );
50       printf( "Card:%3d Suit:%2d Color:%2d\n",
51          wDeck[ k2 ].face, wDeck[ k2 ].suit, wDeck[ k2 ].color );
52    } // конец for
53 } // конец функции deal
```

Пример 10.10 | Вывод программы из примера 10.9

```
Card:  0  Suit: 0  Color: 0  Card:  0  Suit: 2  Color: 1
Card:  1  Suit: 0  Color: 0  Card:  1  Suit: 2  Color: 1
Card:  2  Suit: 0  Color: 0  Card:  2  Suit: 2  Color: 1
Card:  3  Suit: 0  Color: 0  Card:  3  Suit: 2  Color: 1
Card:  4  Suit: 0  Color: 0  Card:  4  Suit: 2  Color: 1
Card:  5  Suit: 0  Color: 0  Card:  5  Suit: 2  Color: 1
Card:  6  Suit: 0  Color: 0  Card:  6  Suit: 2  Color: 1
Card:  7  Suit: 0  Color: 0  Card:  7  Suit: 2  Color: 1
Card:  8  Suit: 0  Color: 0  Card:  8  Suit: 2  Color: 1
Card:  9  Suit: 0  Color: 0  Card:  9  Suit: 2  Color: 1
Card: 10  Suit: 0  Color: 0  Card: 10  Suit: 2  Color: 1
Card: 11  Suit: 0  Color: 0  Card: 11  Suit: 2  Color: 1
Card: 12  Suit: 0  Color: 0  Card: 12  Suit: 2  Color: 1
Card:  0  Suit: 1  Color: 0  Card:  0  Suit: 3  Color: 1
Card:  1  Suit: 1  Color: 0  Card:  1  Suit: 3  Color: 1
Card:  2  Suit: 1  Color: 0  Card:  2  Suit: 3  Color: 1
Card:  3  Suit: 1  Color: 0  Card:  3  Suit: 3  Color: 1
Card:  4  Suit: 1  Color: 0  Card:  4  Suit: 3  Color: 1
Card:  5  Suit: 1  Color: 0  Card:  5  Suit: 3  Color: 1
Card:  6  Suit: 1  Color: 0  Card:  6  Suit: 3  Color: 1
Card:  7  Suit: 1  Color: 0  Card:  7  Suit: 3  Color: 1
Card:  8  Suit: 1  Color: 0  Card:  8  Suit: 3  Color: 1
Card:  9  Suit: 1  Color: 0  Card:  9  Suit: 3  Color: 1
Card: 10  Suit: 1  Color: 0  Card: 10  Suit: 3  Color: 1
Card: 11  Suit: 1  Color: 0  Card: 11  Suit: 3  Color: 1
Card: 12  Suit: 1  Color: 0  Card: 12  Suit: 3  Color: 1
```

Неименованное битовое поле нулевой ширины используется для выравнивания следующих битовых полей *по границам единиц хранения*. Например, в определении структуры:

```
struct example {
   unsigned int a : 13;
   unsigned int   : 0;
   unsigned int   : 4;
}; // конец структуры example
```

используется неименованное битовое поле нулевой ширины, чтобы пропустить оставшиеся незанятыми биты (сколько бы их ни было) в единице хранения, где хранится поле `a`, и выравнивает поле `b` по границе следующей единицы хранения.

Операции с битовыми полями зависят от реализации.

Попытка обращения к отдельным битам битового поля как к элементам массива является синтаксической ошибкой. Битовые поля не являются «массивами битов».

Попытка получить адрес битового поля является синтаксической ошибкой (оператор `&` не может применяться к битовым полям, потому что они не являются адресуемыми элементами).

Несмотря на то что битовые поля позволяют экономить память, для выполнения операций с ними компилятор генерирует более медленный машинный код. Это обусловлено необходимостью выполнения дополнительных операций для доступа к фрагментам адресуемых единиц хранения. Это один из многих примеров экономии памяти за счет снижения скорости выполнения, которые часто можно встретить в программировании.

10.11 Константы-перечисления

Перечисления (коротко рассматривались в разделе 5.11) определяются с помощью ключевого слова `enum` и являются множествами целочисленных **констант перечислений**, представленных идентификаторами. Значения констант в перечислениях начинаются с 0, если явно не указано иное, и каждой следующей константе присваивается значение, больше предыдущего на 1. Например, перечисление

```
enum months {
   JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}; // конец перечисления months
```

создаст новый тип `enum months`, в котором идентификаторы получат последовательные значения от 0 до 11 включительно. Чтобы пронумеровать месяцы числами от 1 до 12, воспользуйтесь следующим перечислением:

```
enum months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}; // конец перечисления months
```

Так как первой константе в предыдущем перечислении явно присвоено значение 1, остальные константы будут получать последовательно увеличивающиеся значения, вплоть до 12. *Идентификаторы* в перечислениях *должны быть уникальными*. Значение каждой константы в перечислении можно установить явно, в определении перечисления, путем присваивания требуемого значения идентификатору.

Одно и то же значение *может* быть присвоено *нескольким* членам перечисления. В примере 10.11 используется переменная `month` типа перечисления в инструкции `for` для вывода названий месяцев года из массива `monthName`. Мы поместили в элемент `monthName[0]` пустую строку `""`. Вы же можете поместить в него такую строку, как `***ERROR***`, чтобы сразу заметить появление логической ошибки.

Попытка присвоить константе перечисления значение после ее определения является синтаксической ошибкой.

Для именования констант в перечислениях используйте только буквы верхнего регистра. Это сделает константы более заметными в тексте программы и будет напоминать, что константы не являются переменными.

Пример 10.11 | Демонстрация использования перечисления

```
// Пример 10.11: fig10_18.c
// Демонстрация использования перечисления
#include <stdio.h>

// перечислить константы, представляющие номера месяцев в году
enum months {
   JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC
}; // конец перечисления months

int main( void )
{
   enum months month; // может хранить номер любого из 12 месяцев

   // инициализировать массив указателей
   const char *monthName[] = { "", "January", "February", "March",
      "April", "May", "June", "July", "August", "September", "October",
      "November", "December" };

   // цикл по месяцам
   for ( ++month ) {
```

```
21       printf( "%2d%11s\n", month, monthName[ month ] );
22    } // конец for
23 } // конец main
```

```
 1 January
 2 February
 3 March
 4 April
 5 May
 6 June
 7 July
 8 August
 9 September
10 October
11 November
12 December
```

10.12 Безопасное программирование на C

Центром CERT было выпущено множество рекомендаций и правил, касающихся тем, обсуждавшихся в данной главе. За дополнительной информацией по каждому из них обращайтесь по адресу: `www.securecoding.cert.org`.

Структуры

Как отмечалось в разделе 10.2.4, требования к выравниванию полей структур могут привести к появлению неиспользуемых байтов в каждой переменной структурного типа. К этой проблеме относятся следующие рекомендации.

- EXP03-C: из-за *требований к выравниванию* размер структуры не всегда является простой суммой размеров составляющих ее полей. Всегда используйте оператор `sizeof` для определения размеров переменных структурных типов. Мы будем использовать этот прием для работы с записями фиксированной длины при записи и чтении файлов в главе 11 и для создания так называемых динамических структур в главе 12.
- EXP04-C: как отмечалось в разделе 10.2.4, структурные переменные не могут сравниваться с помощью операторов равенства или неравенства, потому что они могут содержать байты с неопределенными данными. Поэтому всегда следует сравнивать только отдельные поля структур.
- DCL39-C: в структурной переменной неопределенные дополнительные байты могут содержать секретную информацию, оставшуюся в памяти от предыдущего использования, которая *не* должна оказаться доступной. Данная рекомендация центра CERT описывает специализированные механизмы компиляторов, позволяющие упаковывать данные, устраняя эти дополнительные байты.

typedef

- DCL05-C: объявления сложных типов, таких как указатели на функции, тяжелы для чтения. Используйте `typedef` для создания более простых для чтения имен типов.

Операции с битами

- INT02-C: как результат продвижения правил работы с целыми числами (обсуждаются в разделе 5.6) выполнение поразрядных операций с целочисленными типами меньше, чем `int`, может приводить к неожиданным результатам. Чтобы гарантировать корректность операций, следует использовать явное приведение типов.
- INT13-C: некоторые поразрядные операции со *знаковыми* целочисленными типами возвращают результат, *зависящий от реализации*, – это означает, что одна и та же операция с одними и теми же исходными данными может возвращать разные результаты после компиляции разными компиляторами C. По этой причине для выполнения поразрядных операций всегда используйте *беззнаковые* целочисленные типы.
- EXP17-C: логические операторы `&&` и `||` часто путают с поразрядными операторами `&` и `|` соответственно. Использование операторов `&` и `|` в условной части трехместного оператора (`?:`) может приводить к неожиданным результатам, потому что операторы `&` и `|` не поддерживают вычисления по короткой схеме.

Перечисления

- INT09-C: допустимость присваивания *одного и того же* значения нескольким константам в перечислениях может приводить к сложным в выявлении логическим ошибкам. В большинстве случаев константы перечислений должны иметь *уникальные* значения, чтобы предотвратить подобные логические ошибки.

11

Файлы

В этой главе вы:

- познакомитесь с понятиями файлов и потоков данных;
- узнаете, как читать данные, используя средства последовательного доступа к файлам;
- научитесь читать данные, используя средства произвольного доступа к файлам;
- разработаете программу доступа к файлам.

Краткое содержание

11.1 Введение

В главе 1 мы познакомились с *иерархиями данных*. Данные в переменных и массивах хранятся лишь *временно* – эти данные *теряются* по завершении программы. Для *долговременного* хранения данных используются **файлы**. Компьютеры хранят файлы на таких устройствах, как жесткие диски, компакт-диски, диски DVD и флэш-диски. В этой главе мы узнаем, как создавать, изменять и обрабатывать файлы в программах на языке C. Мы познакомимся с обоими типами файлов – последовательного и произвольного доступа.

11.2 Файлы и потоки данных

С точки зрения языка C, каждый файл является всего лишь последовательным потоком байтов (как показано на рис. 11.1). Каждый файл заканчивается **признаком конца файла** и не может иметь размер больше определенного значения, хранящегося в системных структурах данных. Когда файл *открывается*, в соответствие ему ставится **поток данных**. В каждой программе при запуске открываются три файла и три соответствующих им потока данных – **стандартный ввод**, **стандартный вывод** и **стандартный вывод ошибок**. Потоки являются каналами обмена данными между файлами и программами. Например, поток стандартного ввода позволяет программе читать данные, вводимые с клавиатуры, а поток стандартного вывода – выводить данные на экран. Операция открытия файла возвращает указатель на структуру `FILE` (определена в заголовочном файле `<stdio.h>`), которая содержит информацию, необходимую для работы с файлом. В некоторых операционных системах эта структура включает **дескриптор (описатель) файла**, то есть индекс в **таблице открытых файлов**, поддерживаемой операционной системой. Каждый элемент этой таблицы содержит **блок управления файлом** (File Control Block, FCB), содержащий информацию, необходимую операционной системе для организации управления файлом. Операции со стандартным вводом, стандартным выводом и стандартным выводом ошибок выполняются с помощью указателей на файлы `stdin`, `stdout` и `stderr`.

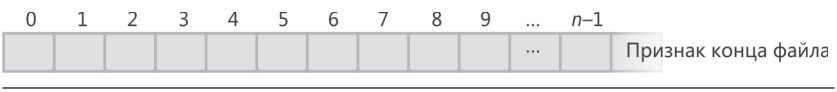


Рис. 11.1 | Файл с точки зрения языка C

Стандартная библиотека содержит множество функций для чтения данных из файлов и записи их в файлы. Функция `fgetc`, своим действием напоминающая функцию `getchar`, читает один символ из файла. Она принимает аргумент с указателем на структуру `FILE` файла, откуда следует прочитать символ. Вызов `fgetc( stdin )` прочитает один символ из потока `stdin` – стандартного ввода, что эквивалентно вызову `getchar()`.

Функция `fputc`, своим действием напоминающая функцию `putchar`, записывает один символ в файл. Она принимает аргумент с указателем на структуру `FILE` файла, куда следует записать символ. Вызов `fputc( 'a', stdout )` запишет символ `'a'` в поток `stdout` – стандартного вывода, что эквивалентно вызову `putchar( 'a' )`.

Существуют и другие функции для работы с файлами, подобные тем, что применяются для чтения данных из стандартного ввода и записи данных в стандартный вывод. Например, функции `fgets` и `fputs` можно использовать для *чтения строки из файла и записи строки в файл* соответственно. В следующих нескольких разделах мы познакомимся с эквивалентами функций `scanf` и `printf`, выполняющими операции чтения и записи с файлами, – `fscanf` и `fprintf`. А затем детально рассмотрим функции `fread` и `fwrite`.

11.3 Создание файла с последовательным доступом

В языке C *отсутствует связь между структурами и файлами*. То есть в самом языке нет понятия *записи в файле*. Тем не менее следующий пример демонстрирует, как организовать хранение своих структур в файле.

Программа в примере 11.1 создает простой *файл с последовательным доступом*, который мог бы использоваться в системе учета задолженностей клиентов. Для каждого клиента программа запрашивает *номер счета*, *имя клиента* и *сумму задолженности* (то есть сумму, которую клиент должен компании за товары и оказанные услуги). Данные о клиенте образуют «запись». Номер счета в данном приложении используется как *ключ записи* – номера счетов будут следовать в файле в порядке возрастания. В данной программе предполагается, что пользователь будет вводить номера счетов по возрастанию. В полноценной системе учета задолженностей можно было бы предусмотреть сортировку записей, чтобы пользователь мог вводить счета в произ-

вольном порядке. [*Обратите внимание*: в примерах 11.2–11.3 используются файлы, созданные программой в примере 11.1, поэтому обязательно выполните пример 11.1, прежде чем приступать к опробованию примеров 11.2–11.2.]

Пример 11.1 | Создание файла с последовательным доступом

```
1 // Пример 11.1: fig11_02.c
2 // Создание файла с последовательным доступом
3 #include <stdio.h>
4
5 int main( void )
6 {
7    unsigned int account; // номер счета
8    char name[ 30 ];      // имя клиента
9    double balance;       // сумма задолженности
10
11   FILE *cfPtr;          // cfPtr = clients.dat указатель на файл
12
13   // открыть файл вызовом fopen, завершить программу в случае ошибки
14   if ( (cfPtr = fopen( "clients.dat", "w") == NULL ) {
15      puts( "File could not be opened" );
16   } // конец if
17   else {
18      puts( "Enter the account, name, and balance." );
19      puts( "Enter EOF to end input." );
20      printf( "%s", "? " );
21      scanf( "%d%29s%lf", &account, name, &balance );
22
23      // записать номер счета, имя клиента и сумму в файл вызовом fprintf
24      while( !feof( stdin ) ){
25         fprintf( cfPtr, "%d %s %.2f\n", account, name, balance );
26         printf( "%s", "? " );
27         scanf( "%d%29s%lf", &account, name, &balance );
28      } // конец while
29
30      fclose( cfPtr ); // закрыть файл вызовом fclose
31   } // конец else
32 } // конец main
```

```
Enter the account, name, and balance.
Enter EOF to end input.
? 100 Jones 24.98
? 200 Doe 345.67
? 300 White 0.00
? 400 Stone -42.16
? 500 Rich 224.62
? ^Z
```

Теперь давайте займемся исследованием программы. В строке 11 создается переменная `cfPtr` – *указатель на структуру* `FILE`. Для каждого отдельного файла в программах на языке C создается своя структура `FILE`. От программиста не требуется знание особенностей структуры `FILE`, чтобы пользоваться файлами, но наиболее любопытные могут взглянуть на ее определение в заголовочном файле `stdio.h`. Чуть ниже будет показано, какими *окольными*

путями структура `FILE` ведет к блоку управления файлом (FCB), поддерживаемому операционной системой.

Для каждого открываемого файла должен объявляться свой указатель на структуру `FILE`. В строке 14 определяется имя файла – `"clients.dat"`, используемого программой для установления связи с файлом. Переменной-указателю `cfPtr` присваивается *адрес структуры* `FILE`, которая создается при открытии файла функцией `fopen`. Функция `fopen` принимает два аргумента: строку с именем файла (которая может включать полный путь к файлу в файловой системе) и строку **режима открытия файла**. Режим `"w"` указывает, что файл открывается для записи (writing). Если для записи открывается несуществующий файл, `fopen` *создаст его*. Если для записи открывается уже существующий файл, его содержимое будет *уничтожено без всяких предупреждений*. Инструкция `if` в программе используется, чтобы проверить значение переменной `cfPtr` на равенство `NULL` (это значение возвращается, если открыть файл не удалось). Если в переменной содержится значение `NULL`, программа выводит сообщение об ошибке и завершается. В противном случае она обрабатывает ввод пользователя и записывает его в файл.

Открытие существующего файла для записи (`"w"`) уничтожит его содержимое, хотя пользователь мог предполагать сохранение этого содержимого.

Попытка использовать неоткрытый файл является логической ошибкой.

Программа предлагает пользователю ввести различные поля для каждой записи или *признак конца файла*. В табл. 11.1 перечислены комбинации клавиш, обеспечивающие ввод этого признака в разных операционных системах.

Таблица 11.1 | Комбинации клавиш для ввода признака конца файла в популярных операционных системах

Операционная система	Комбинация
Linux/Mac OS X/UNIX	*<Ctrl> d*
Windows	*<Ctrl> z*

В строке 24 вызывается функция `feof`, чтобы определить, был ли получен признак конца файла из потока стандартного ввода `stdin`. *Признак конца файла* сообщает программе, что данных для обработки больше нет. В примере 11.1 этот признак устанавливается, когда пользователь завершает ввод нажатием *специальной комбинации клавиш*. В качестве аргумента функции `feof` передается указатель на проверяемый файл (в данном случае `stdin`). Если

признак конца файла установлен, функция вернет ненулевое (истинное) значение; в противном случае она вернет ноль. Инструкция `while`, включающая вызов функции `feof`, продолжает выполняться, пока не будет установлен признак конца файла.

Инструкция в строке 25 выполняет запись данных в файл `clients.dat`. Позднее эти данные смогут быть извлечены из файла другой программой, реализующей чтение файла (см. раздел 11.4). Функция `fprintf` действует подобно функции `printf`, за исключением того, что `fprintf` принимает дополнительный аргумент – указатель на файл, куда выполняется вывод. Эта функция может выводить данные и в поток стандартного вывода, если передать ей `stdout` в качестве указателя на файл, как в следующей инструкции:

```
fprintf( stdout, "%d %s %.2f\n", account, name, balance );
```

После ввода пользователем признака конца файла программа закрывает файл `clients.dat` вызовом функции `fclose` и завершается. Функция `fclose` тоже принимает аргумент с указателем на файл (а не с именем файла). *Если программа не вызовет функцию `fclose` явно, операционная система автоматически закроет файл по завершении программы*. Это один из примеров того, как операционная система «ведет хозяйство».

При закрытии файла освобождаются ресурсы операционной системы, которые могут пригодиться другим пользователям или программам, поэтому всегда следует закрывать файлы сразу же, как только надобность в них отпала, а не надеяться, что операционная система закроет их автоматически по завершении программы.

В листинге, иллюстрирующем сеанс работы с программой, что приводится в примере 11.1, показано, что пользователь ввел информацию по пяти счетам, затем ввел признак конца файла, чтобы сообщить программе об окончании ввода. В примере не показано, как в действительности располагаются введенные данные внутри файла. В следующем разделе демонстрируется пример программы, которая читает данные из файла и выводит их на экран, позволяя тем самым убедиться в успешном создании файла.

На рис. 11.2 иллюстрируются взаимоотношения между указателями на структуры `FILE`, структурами `FILE` и блоками управления файлами (FCB). Когда программа открывает файл `"clients.dat"`, операционная система копирует блок управления файлом в память. Рисунок 11.2 иллюстрирует связь между указателем на файл, возвращаемым функцией `fopen`, и блоком FCB, используемым операционной системой для управления файлом.

Программы могут вообще не использовать файлы, обрабатывать единственный файл или целое множество. Как бы то ни было, но для каждого файла, используемого в программе, должен быть создан отдельный указатель на файл вызовом `fopen`. *Все последующие операции с файлами, выполняемые после их открытия, должны производиться с применением соответствующих указа-*

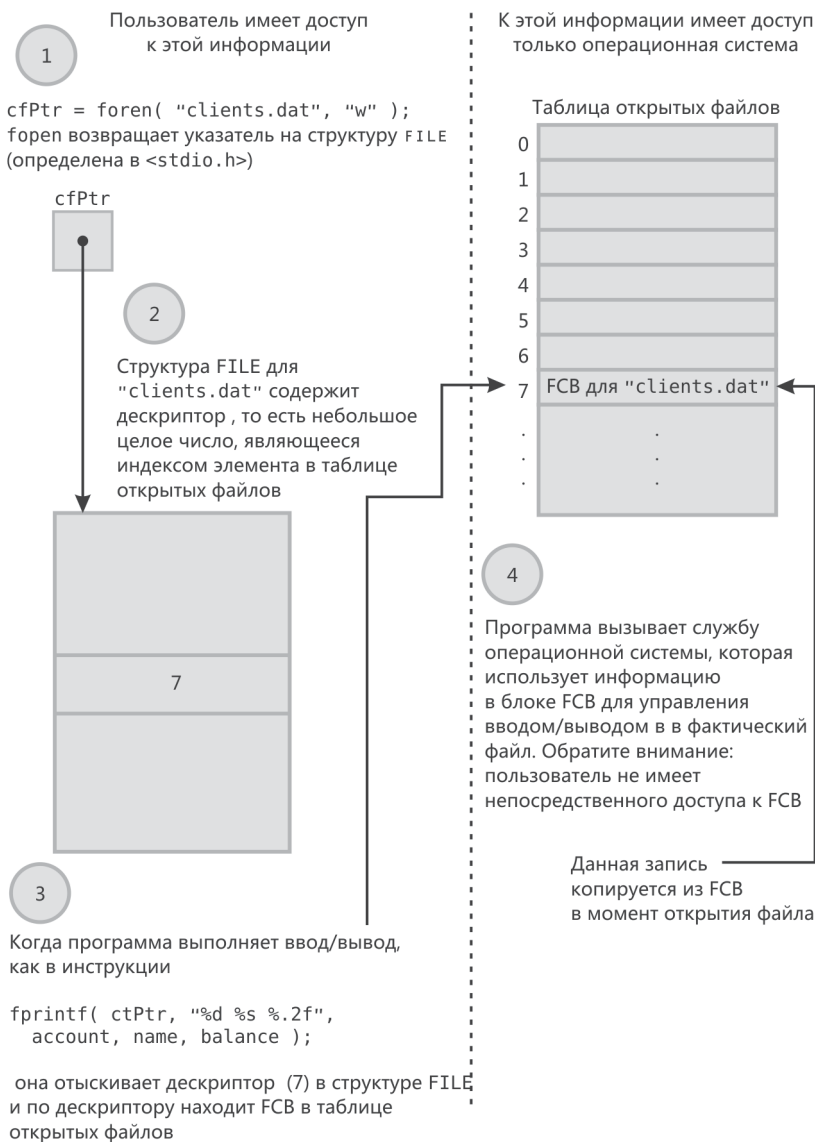


Рис. 11.2 | Взаимоотношения между указателями на структуры FILE, структурами FILE и блоками управления файлами (FCB)

телей на файлы. Файлы могут открываться в одном из множества режимов (табл. 11.2). Чтобы создать новый файл или стереть содержимое существующего файла перед записью, следует указать режим для записи (`"w"`). Чтобы получить возможность прочитать данные из существующего файла, нужно указать режим для чтения (`"r"`). Чтобы получить возможность добавлять новые записи в конец существующего файла, требуется указать режим для добавления в конец (`"a"`). Чтобы получить возможность читать данные из файла и записывать новые данные в файл, следует указать один из трех режимов для обновления – `"r+"`, `"w+"` или `"a+"`. Режим `"r+"` используется с целью открытия существующих файлов для чтения и записи. Режим `"w+"` создаст новый файл, доступный для чтения и записи. Если файл уже существует, его содержимое будет стерто. Режим `"a+"` используется с целью открытия существующих файлов для чтения и записи – все операции записи будут выполняться в конец файла. Если файл не существует, он будет создан. Для всех режимов открытия файлов имеются соответствующие режимы двоичного доступа (содержащие букву `b`), предназначенные для работы с двоичными файлами. Двоичные режимы будут рассматриваться в разделах 11.5–11.9 при описании файлов с произвольным доступом. Кроме того, стандарт C11 предусматривает *исключительный* режим записи, кото-

Таблица 11.2 | Режимы открытия файлов

Режим	Описание
`r`	Открыть существующий файл для чтения
`w`	Создать файл для записи. Если файл уже существует, его содержимое будет стерто
`a`	Добавление в конец (append): открыть или создать файл для записи в конец файла
`r+`	Открыть существующий файл для обновления (чтения и записи)
`w+`	Создать файл для обновления. Если файл уже существует, его содержимое будет стерто
`a+`	Добавление в конец (append): открыть или создать файл для обновления; запись выполняется в конец файла
`rb`	Открыть существующий файл для чтения в двоичном режиме
`wb`	Создать файл для записи в двоичном режиме. Если файл уже существует, его содержимое будет стерто
`ab`	Добавление в конец (append): открыть или создать файл для записи в конец файла в двоичном режиме
`rb+`	Открыть существующий файл для обновления (чтения и записи) в двоичном режиме
`wb+`	Создать файл для обновления в двоичном режиме. Если файл уже существует, его содержимое будет стерто
`ab+`	Добавление в конец (append): открыть или создать файл для обновления в двоичном режиме; запись выполняется в конец файла

рый обозначается добавлением буквы x в конец режимов w, w+, wb и wb+. Если указан исключительный режим, функция fopen будет терпеть неудачу, если требуемый файл уже существует или не может быть создан. Если открытие файла в исключительном режиме увенчалось успехом и операционная система поддерживает механизм исключительного доступа к файлам, тогда *только* ваша программа сможет работать с файлом, пока он открыт. (Некоторые компиляторы и программы не поддерживают исключительного режима для записи.) Если при открытии файла в любом режиме возникнет ошибка, fopen вернет NULL.

Попытка открыть несуществующий файл для чтения приведет к ошибке.

Попытка открыть файл для чтения или для записи, не обладая соответствующими правами доступа к файлу, приведет к ошибке во время выполнения (зависит от операционной системы).

Попытка открыть файл для записи в отсутствие свободного места на диске приведет к ошибке во время выполнения.

Открытие файла в режиме для чтения ("w"), когда он должен быть открыт в режиме для обновления ("r+"), уничтожит содержимое файла.

Открывайте файл только в режиме для чтения (а не обновления), если его содержимое не должно изменяться. Это предотвратит случайные попытки изменить данные в файле. Это – еще один пример применения принципа наименьших привилегий.

11.4 Чтение данных из файла с последовательным доступом

Данные хранятся в файлах так, что позднее они могут быть извлечены из файла для дополнительной обработки. В предыдущем разделе было показано, как создавать файлы с последовательным доступом. В этом разделе демонстрируется возможность чтения данных из файла с последовательным доступом.

Программа в примере 11.2 читает записи из файла "clients.dat", созданные программой в примере 11.1, и выводит их содержимое. В строке 11 создается переменная cfPtr – *указатель на структуру* FILE. В строке 14 выполняется попытка открыть файл "clients.dat" для чтения ("r") и проверяется

успешность операции открытия (функция `fopen` должна вернуть значение, отличное от `NULL`). В строке 19 выполняется чтение «записи» из файла. Функция `fscanf` является эквивалентом функции `scanf`, с той лишь разницей, что `fscanf` принимает в качестве аргумента указатель на файл, откуда следует прочитать данные. Когда эта инструкция выполнится в первый раз, в переменной `account` окажется значение `100`, в переменной `name` – значение `"Jones"` и в переменной `balance` – значение `24.98`. Когда функция `fscanf` будет вызвана во *второй* раз (в строке 24), она прочитает следующую запись из файла и переменные `account`, `name` и `balance` получат новые значения. Когда будет достигнут конец файла, программа закроет его (строка 27) и завершит выполнение. Функция `feof` возвращает истинное значение только *после* попытки прочитать несуществующие данные после достижения конца последней строки в файле.

Пример 11.2 | Чтение и вывод содержимого файла с последовательным доступом

```
 1 // Пример 11.2: fig11_06.c
 2 // Чтение и вывод содержимого файла с последовательным доступом
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    unsigned int account; // номер счета
 8    char name[ 30 ];      // имя клиента
 9    double balance;       // сумма задолженности
10
11    FILE *cfPtr;          // указатель на файл clients.dat
12
13    // открыть файл вызовом fopen, завершить программу в случае ошибки
14    if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
15       puts( "File could not be opened" );
16    } // конец if
17    else { // прочитать номер счета, имя клиента и сумму долга из файла
18       printf( "%-10s%-13s%s\n", "Account", "Name", "Balance" );
19       fscanf( cfPtr, "%d%29s%lf", &account, name, &balance );
20
21       // пока не достигнут конец файла
22       while( !feof( cfPtr ) ){
23          printf( "%-10d%-13s%7.2f\n", account, name, balance );
24          fscanf( cfPtr, "%d%29s%lf", &account, name, &balance );
25       } // конец while
26
27       fclose( cfPtr ); // закрыть файл вызовом fclose
28    } // конец else
29 } // конец main
```

```
Account  Name        Balance
100      Jones         24.98
200      Doe          345.67
300      White          0.00
400      Stone        -42.16
500      Rich         224.62
```

Сброс указателя текущей позиции в файле

При последовательном извлечении данных из файла программа обычно начинает чтение содержимого с самого начала и продолжает читать данные, пока не будет достигнута требуемая запись. Иногда в течение работы программы этот процесс чтения бывает желательно повторить несколько раз (с начала файла). Инструкция

```
rewind( cfPtr );
```

переустанавливает **указатель текущей позиции в файле** – определяющий порядковый номер байта в файле, который будет прочитан или записан, – в *начало* файла (то есть на байт с порядковым номером 0). Указатель текущей позиции в файле в действительности *не* является указателем. Это целое число, определяющее номер байта в файле, который будет прочитан или записан следующим. Иногда порядковый номер байта называют **смещением в файле**. Указатель текущей позиции в файле является полем в структуре `FILE`.

Программа вывода информации о задолженностях

Программа в примере 11.3 позволяет заведующему отдела кредитования получить списки клиентов с нулевой задолженностью (которые ничего не должны компании), отрицательной задолженностью (которым компания должна деньги) и положительной задолженностью (которые должны компании за товары и оказанные услуги). Отрицательная задолженность определяется *отрицательным* значением суммы, положительная – *положительным*.

Пример 11.3 | Программа вывода информации о задолженностях

```
// Пример 11.3: fig11_07.c
// Программа вывода информации о задолженностях
#include <stdio.h>

// выполнение программы начинается с функции main
int main( void )
{
   unsigned int request; // запрошенный номер пункта меню
   unsigned int account; // номер счета
   double balance;       // сумма задолженности
   char name[ 30 ];      // имя клиента
   FILE *cfPtr;          // указатель на файл clients.dat

   // открыть файл вызовом fopen, завершить программу в случае ошибки
   if ( ( cfPtr = fopen( "clients.dat", "r" ) ) == NULL ) {
      puts( "File could not be opened" );
   } // конец if
   else {

      // вывести меню
      printf( "%s", "Enter request\n"
         " 1 - List accounts with zero balances\n"
```

```
23          " 2 - List accounts with credit balances\n"
24          " 3 - List accounts with debit balances\n"
25          " 4 - End of run\n? " );
26       scanf( "%u", &request );
27
28       // обработать запрос пользователя
29       while ( request != 4 ) {
30
31          // прочитать номер, имя и сумму из файла
32          fscanf( cfPtr, "%d%29s%lf", &account, name, &balance );
33
34          switch ( request ) {
35          case 1:
36             puts( "\nAccounts with zero balances:" );
37
38             // читать содержимое файла до конца
39             while ( !feof( cfPtr ) ) {
40
41                if ( balance == 0 ) {
42                   printf( "%-10d%-13s%7.2f\n",
43                      account, name, balance );
44                } // конец if
45
46                // прочитать номер, имя и сумму из файла
47                fscanf( cfPtr, "%d%29s%lf",
48                   &account, name, &balance );
49             } // конец while
50
51             break;
52          case 2:
53             puts( "\nAccounts with credit balances:\n" );
54
55             // читать содержимое файла до конца
56             while ( !feof( cfPtr ) ) {
57
58                if ( balance < 0 ) {
59                   printf( "%-10d%-13s%7.2f\n",
60                      account, name, balance );
61                } // конец if
62
63                // прочитать номер, имя и сумму из файла
64                fscanf( cfPtr, "%d%29s%lf",
65                   &account, name, &balance );
66             } // конец while
67
68             break;
69          case 3:
70             puts( "\nAccounts with debit balances:\n" );
71
72             // читать содержимое файла до конца
73             while ( !feof( cfPtr ) ) {
74
75                if ( balance > 0 ) {
76                   printf( "%-10d%-13s%7.2f\n",
77                      account, name, balance );
78                } // конец if
79
80                // прочитать номер, имя и сумму из файла
```

```
81                fscanf( cfPtr, "%d%29s%lf",
82                   &account, name, &balance );
83             } // конец while
84
85             break;
86          } // конец switch
87
88          rewind( cfPtr ); // вернуться к началу файла
89
90          printf( "%s", "\n? " );
91          scanf( "%d", &request );
92       } // конец while
93
94       puts( "End of run." );
95
96    } // конец else
97 } // конец main
```

Программа выводит меню и позволяет заведующему отдела кредитования ввести номер того или иного пункта меню, чтобы получить соответствующую информацию. При выборе пункта 1 программа выводит список клиентов с нулевой задолженностью. При выборе пункта 2 программа выводит список клиентов с отрицательной задолженностью. И при выборе пункта 3 – с положительной задолженностью. Выбор пункта 4 завершает выполнение программы. Пример сеанса работы с программой приводится в примере 11.4.

Пример 11.4 | Сеанс работы с программой вывода информации о задолженностях

```
Enter request
1 - List accounts with zero balances
2 - List accounts with credit balances
3 - List accounts with debit balances
4 - End of run
? 1

Accounts with zero balances:
300       White           0.00

? 2

Accounts with credit balances:
400       Stone         -42.16

? 3

Accounts with debit balances:
100       Jones          24.98
200       Doe           345.67
500       Rich          224.62

? 4

End of run.
```

Данные в таком файле с последовательным доступом не могут быть изменены без риска повредить другие данные. Например, чтобы изменить имя «White» на «Worthington», новое имя нельзя просто перезаписать поверх прежнего. Запись для клиента White выглядит в файле так:

```
300 White 0.00
```

Если просто записать новую запись поверх прежней, получится запись:

```
300 Worthington 0.00
```

которая длиннее (содержит больше символов) прежней. Символы, следующие за вторым символом «о» в имени «Worthington», *наложатся* на начало следующей записи в файле. Проблема в данном случае заключается в том, что в модели форматированного ввода/вывода, основанной на использовании `fprintf` и `fscanf`, поля (а значит, и записи) могут иметь разный размер. Например, значения 7, 14, –117, 2074 и 27383 являются целыми числами, занимающими во внутреннем представлении одинаковое количество байтов, но на экране и в текстовом файле они имеют разную длину.

По этой причине модель последовательного доступа на основе `fprintf` и `fscanf` обычно *не* используется для *обновления записей на месте* – как правило, в таких случаях файл перезаписывается целиком. Чтобы изменить имя, как было показано выше, записи до `300 White 0.00` следует скопировать в новый файл, записать туда же новую запись и затем скопировать оставшиеся записи. Чтобы изменить одну запись, придется обработать все записи.

11.5 Произвольный доступ к файлам

Как уже отмечалось выше, записи в файле, что создаются с помощью функции `fprintf` форматированного вывода, не всегда имеют одинаковую длину. Но, в отличие от файлов с последовательным доступом, записи в файлах с **произвольным доступом** имеют *фиксированный размер* и к ним можно обращаться непосредственно (а значит, быстро), без необходимости читать все предыдущие записи. Это обеспечивает более высокую пригодность файлов с произвольным доступом для использования в системах резервирования авиабилетов, банковских системах, системах электронной торговли и других подобных системах, требующих быстрого доступа к определенным данным. Существует множество разных способов организации файлов с произвольным доступом, но мы ограничимся обсуждением самого простого подхода, основанного на использовании записей фиксированного размера.

Так как все записи в файле с произвольным доступом имеют одинаковый размер, точное местоположение записи относительно начала файла вычисляется как функция от значения ключа. Вскоре вы увидите, насколько сильно это упрощает непосредственный доступ к записям даже в очень больших файлах.

На рис. 11.3 представлен один из способов организации файла с произвольным доступом. Такой файл напоминает железнодорожный товарный состав со множеством вагонов: некоторые из них могут быть пустыми, другие – полными. Все вагоны в таком составе имеют одинаковую емкость.

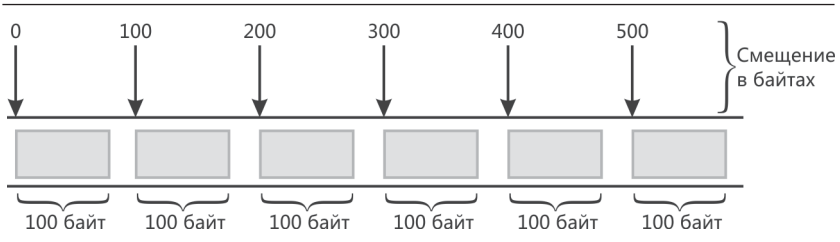


Рис. 11.3 | Организация файла с произвольным доступом

Фиксированный размер записей позволяет вставлять данные в файл с произвольным доступом в любую запись, *не опасаясь затереть данные в соседних записях*. Данные, сохраненные прежде, также можно изменять или стирать, не перезаписывая файла целиком. В следующих разделах рассказывается, как создать файл с произвольным доступом, как вводить в него данные, как читать данные последовательно или выборочно, как изменять данные и как удалять их.

11.6 Создание файла с произвольным доступом

Функция `fwrite` выводит указанное количество байтов из указанной области памяти в указанный файл. Запись в файл выполняется, начиная со смещения, определяемого указателем текущей позиции в файле. Функция `fread` читает указанное количество байтов из файла, начиная со смещения, определяемого указателем текущей позиции в файле, в указанную область памяти. Теперь целое число можно записать не как последовательность символов цифр:

```
fprintf( fPtr, "%d", number );
```

которая может состоять и из одного, и из одиннадцати символов (10 цифр плюс знак числа), а как 4-байтное целое значение:

```
fwrite( &number, sizeof( int ), 1, fPtr );
```

которое всегда будет иметь размер 4 байта в системе, где целые числа имеют 4-байтное двоичное представление. Инструкция выше выведет 4-байтное значение переменной `number` в файл, представленный указателем `fPtr` (на-

значение аргумента 1 будет разъяснено чуть ниже). Позднее мы сможем прочитать эти четыре байта в переменную number с помощью функции fread. Функции fread и fwrite читают и записывают данные, такие как целые числа, фиксированного размера, используя двоичное их представление, а не текстовое, как функции printf и scanf, доступное для человека. Из-за того, что двоичное представление тесно связано с аппаратной архитектурой компьютера, файлы с такими данными могут оказаться несовместимыми с другими системами или программами, скомпилированными другими компиляторами или с иными флагами компиляции.

Функции fwrite и fread поддерживают возможность записи массивов данных в файлы и чтения их из файлов. Третий аргумент обеих функций, fread и fwrite, – это количество элементов массива, которое следует прочитать из файла или записать в файл. Предыдущий пример вызова функции fwrite записывает в файл единственное целое число, поэтому в третьем аргументе передается число 1 (то есть функции предписывается вывести один элемент массива).

Программы редко записывают в файлы значения по одному. Обычно запись выполняется структурами, как будет показано в следующих примерах.

Ознакомьтесь с постановкой задачи ниже:

> *Создайте систему обработки задолженностей, способную хранить до 100 записей фиксированного размера. Каждая запись должна включать номер счета, который будет использоваться как ключ записи, имя и фамилию клиента и сумму задолженности. Программа должна позволять обновлять информацию о счете, вставлять новые записи, удалять записи и выводить список всех записей в текстовый файл, готовый для печати на принтере. Для реализации системы используйте файл с произвольным доступом.*

В следующих нескольких разделах будут представлены приемы работы с файлами, которые пригодятся при создании программы обработки задолженностей. Программа в примере 11.5 демонстрирует, как открыть файл с произвольным доступом, как определить формат записи с использованием структуры, как записать данные в файл и как закрыть его. Эта программа инициализирует все 100 записей в файле "credit.dat" пустыми структурами с помощью функции fwrite. Каждая пустая структура содержит 0 в поле номера счета, пустую строку "" в поле имени, пустую строку "" в поле фамилии и 0.0 в поле с суммой задолженности. Инициализация файла выполняется с целью зарезервировать место на диске для хранения файла.

Пример 11.5 | Создание файла с произвольным доступом

```
1 // Пример 11.5: fig11_10.c
2 // Создание файла с произвольным доступом
3 #include <stdio.h>
4
5 // определение структуры clientData
6 struct clientData {
7    unsigned int acctNum; // номер счета
```

```
 8    char lastName[ 15 ];  // фамилия клиента
 9    char firstName[ 10 ]; // имя клиента
10    double balance;       // сумма задолженности
11 }; // конец структуры clientData
12
13 int main( void )
14 {
15    unsigned int i; // счетчик для подсчета итераций 1-100
16
17    // создать экземпляр clientData с информацией по умолчанию
18    struct clientData blankClient = { 0, "", "", 0.0 };
19
20    FILE *cfPtr;    // указатель на файл credit.dat
21
22    // открыть файл вызовом fopen, завершить программу в случае ошибки
23    if ( ( cfPtr = fopen( "credit.dat", "wb" ) ) == NULL ) {
24       puts( "File could not be opened." );
25    } // конец if
26    else {
27       // вывести 100 пустых записей в файл
28       for ( i = 1; i <= 100; ++i ) {
29          fwrite( &blankClient, sizeof( struct clientData ), 1, cfPtr );
30       } // конец for
31
32       fclose ( cfPtr ); // закрыть файл вызовом fclose
33    } // конец else
34 } // конец main
```

Функция `fwrite` записывает блок байтов в файл. Инструкция в строке 29 производит запись структуры `blankClient`, имеющей размер `sizeof(struct clientData)`, в файл, на который ссылается переменная-указатель `cfPtr`. Оператор `sizeof` возвращает размер в байтах операнда, заключенного в круглые скобки (в данном случае – тип `struct clientData`).

Фактически функцию `fwrite` можно использовать для записи сразу нескольких элементов массива объектов. Для этого следует передать функции `fwrite` указатель на массив в первом аргументе и количество элементов для записи – в третьем. В предыдущей инструкции `fwrite` используется для записи единственного объекта, не являющегося элементом массива. Запись одного объекта эквивалентна записи одного элемента массива, поэтому в третьем аргументе функции `fwrite` передано значение `1`. [*Обратите внимание*: в примерах 11.6, 11.8 и 11.9 используется файл данных, созданный в примере 11.5, поэтому выполните пример 11.5, прежде чем опробовать примеры 11.6, 11.8 и 11.9.]

11.7 Запись данных в файл с произвольным доступом

Программа в примере 11.6 записывает данные в файл `"credit.dat"`. Для сохранения данных в определенной позиции в файле она использует комбинацию вызовов функций `fseek` и `fwrite`. Функция `fseek` устанавливает

указатель текущей позиции в файле в заданное смещение, а `fwrite` записывает данные. Демонстрация сеанса работы с программой приводится в примере 11.7.

Пример 11.6 | Запись данных в файл с произвольным доступом

```
// Пример 11.6: fig11_11.c
// Запись данных в файл с произвольным доступом
#include <stdio.h>

// определение структуры clientData
struct clientData {
   unsigned int acctNum; // номер счета
   char lastName[ 15 ];  // фамилия клиента
   char firstName[ 10 ]; // имя клиента
   double balance;       // сумма задолженности
}; // конец структуры clientData

int main( void )
{
   FILE *cfPtr; // указатель на файл credit.dat

   // создать экземпляр clientData с информацией по умолчанию
   struct clientData blankClient = { 0, "", "", 0.0 };

   // открыть файл вызовом fopen, завершить программу в случае ошибки
   if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
      puts( "File could not be opened." );
   } // конец if
   else {
      // предложить пользователю указать номер счета
      printf( "%s", "Enter account number"
         " ( 1 to 100, 0 to end input )\n? " );
         scanf( "%d", &client.acctNum );

      // пользователь вводит информацию, которая копируется в файл
      while ( client.acctNum != 0 ) {
         // предложить ввести фамилию, имя и задолженность
         printf( "%s", "Enter lastname, firstname, balance\n? " );

         // установить значения полей lastName, firstName и balance
         fscanf( stdin, "%14s%9s%lf", client.lastName,
            client.firstName, &client.balance )

         // перейти к указанной пользователем позиции в файле
         fseek( cfPtr, ( client.acctNum - 1 ) *
            sizeof( struct clientData ), SEEK_SET );

         // записать указанную пользователем информацию в файл
         fwrite( &client, sizeof( struct clientData ), 1, cfPtr );

         // предложить пользователю ввести другой номер счета
         printf( "%s", "Enter account number\n? " );
         scanf( "%d", &client.acctNum );
      } // конец while

      fclose( cfPtr ); // закрыть файл вызовом fclose
   } // конец else
} // конец main
```

Строки 40–41 устанавливают указатель текущей позиции в файле `cfPtr` в значение, определяемое выражением `(client.accountNum - 1) * sizeof(struct clientData)`. Значение этого выражения называется **смещением**. Так как номера счетов находятся в диапазоне от 1 до 100, а нумерация байтов в файле начинается с `0`, мы вычитаем 1 из номера счета при вычислении местоположения записи. То есть первая запись будут располагаться в файле, начиная с нулевого байта. Символическая константа `SEEK_SET` указывает, что значение для позиционирования указателя текущей позиции в файле должно откладываться от начала файла на указанное количество байтов. Как видно из инструкции выше, операция позиционирования указателя текущей позиции в файле для первой записи установит его в начало файла, потому что выражение вернет значение `0`. Рисунок 11.4 иллюстрирует файловый указатель, ссылающийся на структуру `FILE`. Указатель текущей позиции в файле на этой диаграмме отражает номер байта, с которого начнется выполнение следующей операции чтения или записи.

Пример 11.7 | Сеанс работы с программой из примера 11.6

```
Enter account number ( 1 to 100, 0 to end input )
? 37
Enter lastname, firstname, balance
? Barker Doug 0.00
Enter account number
? 29
Enter lastname, firstname, balance
? Brown Nancy -24.54
Enter account number
? 96
Enter lastname, firstname, balance
? Stone Sam 34.98
Enter account number
? 88
Enter lastname, firstname, balance
? Smith Dave 258.34
Enter account number
? 33
Enter lastname, firstname, balance
? Dunn Stacey 314.33
Enter account number
? 0
```

Функция `fseek` имеет следующий прототип:

```
int fseek( FILE *stream, long int offset, int whence );
```

где `offset` – количество байтов, на которое нужно сместиться относительно точки отсчета `whence` в файле, определяемом указателем `stream`: положительное значение `offset` соответствует смещению вперед, отрицательное – назад. Аргумент `whence` может принимать одно из трех значений, `SEEK_SET`, `SEEK_CUR` или `SEEK_END` (все определены в заголовочном файле `<stdio.h>`),

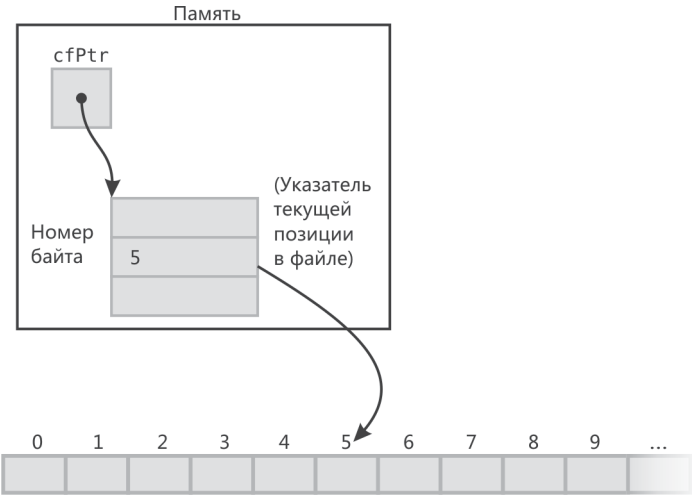


Рис. 11.4 | Указатель текущей позиции в файле ссылается на пятый байт от начала файла

определяющее точку отсчета смещения. `SEEK_SET` – смещение откладывается от *начала* файла; `SEEK_CUR` – от *текущей позиции* в файле; `SEEK_END` – от *конца* файла.

Для простоты программы в этой главе не будут проверять наличие ошибок. Однако в промышленных программах обязательно следует контролировать правильность работы таких функций, как `fscanf` (строки 36–37), `fseek` (строки 40–41) и `fwrite` (строка 44), проверяя возвращаемые ими значения. Функция `fscanf` возвращает количество успешно прочитанных элементов данных или значение `EOF`, если при чтении возникла проблема. Функция `fseek` возвращает ненулевое значение, если операция перемещения указателя текущей позиции не может быть выполнена. Функция `fwrite` возвращает количество элементов, успешно записанных в файл. Если это число меньше величины третьего аргумента, следовательно, во время записи произошла ошибка.

11.8 Чтение данных из файла с произвольным доступом

Функция `fread` читает указанное количество байтов из файла в память. Например, инструкция

```
fread( &client, sizeof( struct clientData ), 1, cfPtr );
```

прочитает количество байтов, определяемое выражением `sizeof(struct clientData)` из файла, на который ссылается `cfPtr`, сохранит их в `client` и вернет количество прочитанных байтов. Чтение байтов начинается в файле со смещения, определяемого указателем текущей позиции в файле. Функция `fread` может прочитать сразу несколько элементов массива, для чего ей следует передать указатель на массив в первом аргументе и количество элементов для чтения – в третьем. Функция `fread` возвращает количество успешно прочитанных элементов. Если это число меньше величины в третьем аргументе, следовательно, во время чтения возникла ошибка.

Программа в примере 11.8 последовательно читает все записи в файле `"credit.dat"`, определяет заполненные записи и выводит хранящиеся в них данные на экран. Функция `feof` помогает определить достижение конца файла, а функция `fread` перемещает очередную порцию данных из файла в переменную `client` структурного типа `clientData`.

Пример 11.8 | Последовательное чтение данных из файла с произвольным доступом

```
// Пример 11.8: fig11_14.c
// Последовательное чтение данных из файла с произвольным доступом
#include <stdio.h>

// определение структуры clientData
struct clientData {
   unsigned int acctNum; // номер счета
   char lastName[ 15 ];  // фамилия клиента
   char firstName[ 10 ]; // имя клиента
   double balance;       // сумма задолженности
}; // конец структуры clientData

int main( void )
{
   FILE *cfPtr; // указатель на файл credit.dat
   int result;  // использует для проверки значения, возвращаемого fread

   // создать экземпляр clientData с информацией по умолчанию
   struct clientData client = { 0, "", "", 0.0 };

   // открыть файл вызовом fopen, завершить программу в случае ошибки
   if ( ( cfPtr = fopen( "credit.dat", "rb" ) ) == NULL ) {
      puts( "File could not be opened." );
   } // конец if
   else {
      printf( "%-6s%-16s%-11s%10s\n", "Acct", "Last Name",
         "First Name", "Balance" );

      // прочитать все записи из файла
      while ( !feof( cfPtr ) ) {
         result = fread( &client, sizeof( struct clientData ), 1, cfPtr );

         // вывести запись
         if ( result != 0 && client.acctNum != 0 ) {
```

```
35             printf( "%-6d%-16s%-11s%10.2f\n",
36                client.acctNum, client.lastName,
37                client.firstName, client.balance );
38          } // конец if
39       } // конец while
40
41       fclose( cfPtr ); // закрыть файл вызовом fclose
42    } // конец else
43 } // конец main
```

```
Acct  Last Name     First Name   Balance
29    Brown         Nancy         -24.54
33    Dunn          Stacey        314.33
37    Barker        Doug            0.00
88    Smith         Dave          258.34
96    Stone         Sam            34.98
```

11.9 Пример: реализация программы для работы со счетами

Теперь рассмотрим реализацию более серьезной программы для работы с файлами (пример 11.9). Программа обслуживает информацию о счетах клиентов – обновляет имеющуюся информацию, добавляет новые и удаляет ненужные счета, а также сохраняет список всех счетов в текстовом файле. Далее предполагается, что предварительно был выполнен пример 11.5 и файл `credit.dat` был успешно создан.

Программа предлагает на выбор пять пунктов меню. При выборе пункта 1 вызывается функция `textFile` (строки 63–94), сохраняющая список всех счетов (обычно такой список называют отчетом) в текстовый файл с именем `accounts.txt`. Эта функция использует `fread` и реализует прием последовательного доступа, воплощенный в программе из примера 11.8. После выбора пункта 1 файл `accounts.txt` будет содержать:

```
Acct  Last Name     First Name   Balance
29    Brown         Nancy         -24.54
33    Dunn          Stacey        314.33
37    Barker        Doug            0.00
88    Smith         Dave          258.34
96    Stone         Sam            34.98
```

При выборе пункта 2 вызывается функция `updateRecord` (строки 97–141), обновляющая информацию о счете в файле. Функция обновляет только уже существующие записи, поэтому она сначала проверяет наличие указанной пользователем записи. Запись читается из файла в структуру `client` вызовом `fread`, и затем поле `acctNum` сравнивается со значением 0. Если оно равно 0, запись интерпретируется как пустая и на экран выводится сообщение об этом. После этого на экран вновь выводятся пункты меню. Если запись содержит значимую информацию, функция `updateRecord` вводит ве-

личину транзакции, вычисляет новую сумму задолженности и записывает запись обратно в файл. Ниже представлен типичный вывод программы при выборе пункта 2:

```
Enter account to update ( 1 - 100 ): 37
37    Barker          Doug             0.00

Enter charge ( + ) or payment ( - ): +87.99
37    Barker          Doug            87.99
```

При выборе пункта 3 вызывается функция `newRecord` (строки 178–217), добавляющая новую запись в файл. Если пользователь введет номер уже существующего счета, `newRecord` выведет сообщение об ошибке и вновь отобразит пункты меню. Эта функция действует точно так же, как программа в примере 11.6. Ниже представлен типичный вывод программы при выборе пункта 3:

```
Enter new account number ( 1 - 100 ): 22
Enter lastname, firstname, balance
? Johnston Sarah 247.45
```

При выборе пункта 4 вызывается функция `deleteRecord` (строки 144–175), удаляющая запись из файла. Перед удалением у пользователя запрашивается номер удаляемого счета и на место соответствующей записи в файле сохраняется пустая структура. Если указанный пользователем счет не содержит информации, `deleteRecord` выводит сообщение об ошибке. Пункт 5 завершает выполнение программы. Исходный код программы приводится в примере 11.9. Файл `"credit.dat"` открывается для обновления (чтения и записи) с использованием строки режима `"rb+"`.

Пример 11.9 | Программа для работы с банковскими счетами

```
 1 // Пример 11.9: fig11_15.c
 2 // Банковская программа для работы со счетами. Реализует
 3 // последовательное чтение файла с произвольным доступом, обновляет
 4 // его содержимое, создает новые записи в файле и удаляет существующие.
 5 #include <stdio.h>
 6
 7 // определение структуры clientData
 8 struct clientData {
 9    unsigned int acctNum; // номер счета
10    char lastName[ 15 ];  // фамилия клиента
11    char firstName[ 10 ];  // имя клиента
12    double balance;       // сумма задолженности
13 }; // конец структуры clientData
14
15 // прототипы
16 unsigned int enterChoice( void );
17 void textFile( FILE *readPtr );
18 void updateRecord( FILE *fPtr );
19 void newRecord( FILE *fPtr );
20 void deleteRecord( FILE *fPtr );
21
```

```
22 int main( void )
23 {
24    FILE *cfPtr;         // указатель на файл credit.dat
25    unsigned int choice; // выбор пользователя
26
27    // открыть файл вызовом fopen, завершить программу в случае ошибки
28    if ( ( cfPtr = fopen( "credit.dat", "rb+" ) ) == NULL ) {
29       puts( "File could not be opened." );
30    } // конец if
31    else {
32       // дать пользователю возможность сделать выбор
33       while ( ( choice = enterChoice() ) != 5 ) {
34          switch ( choice ) {
35             // создать текстовый файл из файла с записями
36             case 1:
37                textFile( cfPtr );
38                break;
39             // обновить запись
40             case 2:
41                updateRecord( cfPtr );
42                break;
43             // создать запись
44             case 3:
45                newRecord( cfPtr );
46                break;
47             // удалить существующую запись
48             case 4:
49                deleteRecord( cfPtr );
50                break;
51             // вывести сообщение в случае недопустимого выбора
52             default:
53                puts( "Incorrect choice" );
54                break;
55          } // конец switch
56       } // конец while
57
58       fclose( cfPtr ); // закрыть файл вызовом fclose
59    } // конец else
60 } // конец main
61
62 // создает текстовый файл
63 void textFile( FILE *readPtr )
64 {
65    FILE *writePtr; // указатель на файл accounts.txt
66    int result;     // используется для проверки результата fread
67
68    // создать структуру clientData с информацией по умолчанию
69    struct clientData client = { 0, "", "", 0.0 };
70
71    // открыть файл вызовом fopen, завершить программу в случае ошибки
72    if ( ( writePtr = fopen( "accounts.txt", "w" ) ) == NULL ) {
73       puts( "File could not be opened." );
74    } // конец if
75    else {
76       rewind( readPtr ); // установить текущую позицию в начало файла
77       fprintf( writePtr, "%-6s%-16s%-11s%10s\n",
78          "Acct", "Last Name", "First Name", "Balance" );
```

```
79
80       // скопировать все непустые записи в текстовый файл
81       while ( !feof( readPtr ) ) {
82          result = fread(&client, sizeof( struct clientData ), 1, readPtr);
83
84          // записать одну запись в текстовый файл
85          if ( result != 0 && client.acctNum != 0 ) {
86             fprintf( writePtr, "%-6d%-16s%-11s%10.2f\n",
87                client.acctNum, client.lastName,
88                client.firstName, client.balance );
89          } // конец if
90       } // конец while
91
92       fclose( writePtr ); // закрыть файл вызовом fclose
93    } // конец else
94 } // конец функции textFile
95
96 // обновляет сумму задолженности в записи
97 void updateRecord( FILE *fPtr )
98 {
99    unsigned int account; // номер счета
100   double transaction;   // сумма транзакции
101
102   // создать структуру clientData с информацией по умолчанию
103   struct clientData client = { 0, "", "", 0.0 };
104
105   // получить номер счета у пользователя
106   printf( "%s", "Enter account to update ( 1 - 100 ): " );
107   scanf( "%d", &account );
108
109   // переместить текущую позицию в файле
110   fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
111      SEEK_SET );
112
113   // прочитать запись из файла
114   fread( &client, sizeof( struct clientData ), 1, fPtr );
115
116   // сообщить об ошибке, если такой счет отсутствует
117   if ( client.acctNum == 0 ) {
118      printf( "Account #%d has no information.\n", account );
119   } // конец if
120   else { // обновить запись
121      printf( "%-6d%-16s%-11s%10.2f\n\n",
122         client.acctNum, client.lastName,
123         client.firstName, client.balance );
124
125      // запросить у пользователя сумму транзакции
126      printf( "%s", "Enter charge ( + ) or payment ( - ): " );
127      scanf( "%lf", &transaction );
128      client.balance += transaction; // обновить сумму задолженности
129
130      printf( "%-6d%-16s%-11s%10.2f\n",
131         client.acctNum, client.lastName,
132         client.firstName, client.balance );
133
134      // переместить текущую позицию в файле
135      fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
```

```
136          SEEK_SET );
137
138       // записать обновленную запись в файл поверх старой
139       fwrite( &client, sizeof( struct clientData ), 1, fPtr );
140    } // конец else
141 } // конец функции updateRecord
142
143 // удаляет существующую запись
144 void deleteRecord( FILE *fPtr )
145 {
146    struct clientData client; // хранит запись, прочитанную из файла
147    struct clientData blankClient = { 0, "", "", 0 }; // пустая запись
148
149    unsigned int accountNum; // номер счета
150
151    // Запросить номер удаляемого счета
152    printf( "%s", "Enter account number to delete ( 1 - 100 ): " );
153    scanf( "%d", &accountNum );
154
155    // переместить текущую позицию в файле
156    fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
157       SEEK_SET );
158
159    // прочитать запись из файла
160    fread( &client, sizeof( struct clientData ), 1, fPtr );
161
162    // сообщить об ошибке, если запись не существует
163    if ( client.acctNum == 0 ) {
164       printf( "Account %d does not exist.\n", accountNum );
165    } // конец if
166    else { // удалить запись
167    // переместить текущую позицию в файле
168       fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
169          SEEK_SET );
170
171       // заменить существующую запись пустой
172       fwrite( &blankClient,
173          sizeof( struct clientData ), 1, fPtr );
174    } // конец else
175 } // конец функции deleteRecord
176
177 // создает и вставляет новую запись
178 void newRecord( FILE *fPtr )
179 {
180    // создать структуру clientData с информацией по умолчанию
181    struct clientData client = { 0, "", "", 0.0 };
182
183    unsigned int accountNum; // номер счета
184
185    // получить номер счета для создания
186    printf( "%s", "Enter new account number ( 1 - 100 ): " );
187    scanf( "%d", &accountNum );
188
189    // переместить текущую позицию в файле
190    fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
191       SEEK_SET );
192
```

```
193    // прочитать запись из файла
194    fread( &client, sizeof( struct clientData ), 1, fPtr );
195
196    // сообщить об ошибке, если счет уже существует
197    if ( client.acctNum != 0 ) {
198       printf( "Account #%d already contains information.\n",
199          client.acctNum );
200    } // конец if
201    else { // создать запись
202       // запросить у пользователя имя, фамилию и сумму задолженности
203       printf( "%s", "Enter lastname, firstname, balance\n? " );
204       scanf( "%14s%9s%lf", &client.lastName, &client.firstName,
205          &client.balance );
206
207       client.acctNum = accountNum;
208
209       // переместить текущую позицию в файле
210       fseek( fPtr, ( account - 1 ) * sizeof( struct clientData ),
211          SEEK_SET );
212
213       // вставить запись в файл
214       fwrite( &client,
215          sizeof( struct clientData ), 1, fPtr );
216    } // конец else
217 } // конец функции newRecord
218
219 // вводит выбранный пользователем пункт меню
220 unsigned int enterChoice( void )
221 {
222    unsigned int menuChoice; // переменная для хранения выбора пользователя
223
224    // вывести доступные варианты для выбора
225    printf( "%s", "\nEnter your choice\n"
226       "1 - store a formatted text file of accounts called\n"
227       "    \"accounts.txt\" for printing\n"
228       "2 - update an account\n"
229       "3 - add a new account\n"
230       "4 - delete an account\n"
231       "5 - end program\n? " );
232
233    scanf( "%u", &menuChoice ); // читает выбор пользователя
234    return menuChoice;
235 } // конец функции enterChoice
```

11.10 Безопасное программирование на C

Функции fprintf_s и fscanf_s

Примеры в разделах 11.3–11.4 использовали функции `fprintf` и `fscanf` для записи текста в файл и чтения текста из файла соответственно. Приложение Annex K к стандарту C11 определяет более безопасные версии этих функций с именами `fprintf_s` и `fscanf_s`, идентичные функциям `printf_s` и `scanf_s`, представленным ранее, за исключением того, что принимают в первом аргументе указатель на структуру `FILE`. Если ваша стандартная библиотека C включает эти функции, используйте их вместо `fprintf` и `fscanf`.

Глава 9 стандарта «CERT Secure C Coding Standard»

Глава 9 стандарта «CERT Secure C Coding Standard» посвящена рекомендациям по реализации ввода/вывода – многие из них относятся к обработке файлов в целом, но есть некоторые, которые касаются использования функций для работы с файлами, представленными в данной главе. За дополнительной информацией обращайтесь по адресу: `www.securecoding.cert.org`.

- FIO03-C: если существующий файл открывается для записи с использованием неисключительных режимов открытия (табл. 11.2), функция `fopen` откроет его и сотрет его содержимое, ничем не показывая, что файл существовал до ее вызова. Чтобы гарантировать, что существующий файл *не* будет открыт и уничтожен, можно использовать новый *исключительный режим*, появившийся в C11 (обсуждается в разделе 11.3), позволяющий функции `fopen` открыть файл, *только* если он *не* существует.
- FIO04-C: в промышленном коде всегда следует проверять значения, возвращаемые функциями, которые выполняют операции с файлами, чтобы убедиться, что в процессе их выполнения не возникло ошибок.
- FIO07-C: функция `rewind` не возвращает значения, из-за чего вы не сможете убедиться в успехе операции. Поэтому вместо нее рекомендуется использовать функцию `fseek`, возвращающую ненулевое значение в случае ошибки.
- FIO09-C: в этой главе были продемонстрированы файлы обоих типов, текстовые и двоичные. Из-за разницы представления двоичных данных разными платформами двоичные файлы, записанные в одной платформе, часто оказываются непереносимыми на другие платформы. Чтобы добиться большей переносимости, рассматривайте возможность использования текстовых файлов или реализуйте библиотечные функции, способные учитывать различия в представлении двоичных данных на разных платформах.
- FIO14-C: некоторые библиотечные функции по-разному работают с текстовыми и двоичными файлами. В частности, функция `fseek` не гарантирует правильную работу с двоичными файлами, если в аргументе `whence` ей передается `SEEK_END`, поэтому старайтесь всегда использовать `SEEK_SET`.
- FIO42-C: многие платформы ограничивают количество файлов, которые программа может держать открытыми одновременно. Поэтому всегда закрывайте файлы, как только надобность в них отпадает.

12

Структуры данных

В этой главе вы:

- узнаете, как динамически выделять и освобождать память для объектов данных;
- познакомитесь со связыванием структур с помощью указателей, со структурами, ссылающимися на самих себя, и с рекурсией;
- научитесь создавать и управлять связанными списками, очередями, стеками и двоичными деревьями;
- познакомитесь с важнейшими областями применения связанных структур данных.

Краткое содержание

12.1 Введение

В предыдущих главах мы познакомились со структурами данных, имеющими фиксированный размер, такими как одномерные массивы, двумерные массивы и структуры. В этой главе мы познакомимся с динамическими структурами данных, которые могут изменяться в размерах в процессе выполнения программы.

- **Связанные списки** – это коллекции элементов данных, «связанных в цепочку»; вставка и удаление элементов могут выполняться *в любой части* списка.
- **Стеки** – важнейшие структуры данных для компиляторов и операционных систем; вставка и удаление элементов могут выполняться *только с одного конца* стека его **вершине**.
- **Очереди** представляют собой аналог очереди, например, в магазине; вставка элементов может производиться *только в конец* очереди (который также называют **хвостом**), а удаление – *только из начала* очереди (которое также называют **головой**).
- **Двоичные деревья** – структуры, обеспечивающие быстрый поиск и сортировку данных, а также надежное устранение дубликатов элементов данных; деревья используются для представления структур каталогов в файловых системах и компиляции выражений на машинный язык.

Каждая из этих структур имеет множество практических применений.

Мы обсудим каждую структуру данных и реализуем программы, создающие их и выполняющие операции с ними.

12.2 Структуры, ссылающиеся на себя самих

Выше уже говорилось, что структурами данных, ссылающимися на себя самих, называются такие структуры, которые содержат поле-указатель на структуру того же типа. Например, следующее определение:

```
struct node {
   int data;
   struct node *nextPtr;
}; // конец структуры node
```

объявляет тип `struct node`. Структура типа `struct node` имеет два поля – целочисленное поле `data` и поле-указатель `nextPtr`. Поле `nextPtr` указывает на структуру типа `struct node` – *того же самого* типа, что и сама структура, содержащее это поле, отсюда и название – «структуры, ссылающиеся на себя самих». Поля, подобные полю `nextPtr`, называют полями **связи** – они могут использоваться для «связывания» структуры типа `struct node` с другой структурой того же типа. Структуры, ссылающиеся на себя самих, могут *связываться* воедино, образуя другие ценные структуры данных, такие как списки, очереди, стеки и деревья. На рис. 12.1 изображены две структуры, ссылающиеся на себя самих, связанные в список. Обратная косая черта в данном случае представляет нулевой указатель `NULL` – он помещен в поле связи второй структуры, чтобы показать, что она является последней в списке и не указывает ни на какую другую структуру. [*Обратите внимание*: обратная косая черта здесь используется только для иллюстрации; она не имеет ничего общего с символом обратного слэша.] Нулевой указатель `NULL` обычно служит признаком конца структуры данных, так же как и нулевой символ `"\0"` служит признаком конца строки.

Если забыть присвоить полю связи последней структуры в списке значение `NULL`*, это может привести к ошибке во время выполнения.*

Рис. 12.1 | Структуры, ссылающиеся на себя самих, связанные в список

12.3 Динамическое выделение памяти

Создание и манипулирование динамическими структурами данных требует **динамического выделения памяти** – возможности *получать дополнительную память во время выполнения* с целью хранения в ней новых объектов данных и *освобождать эту память, когда надобность в ней исчезает.*

Основными инструментами динамического выделения памяти являются функции `malloc` и `free` и оператор `sizeof`. Функция `malloc` принимает в качестве аргумента количество байтов памяти, которую требуется выделить, и возвращает указатель типа `void *` на выделенный блок памяти. Как вы наверняка помните, указатель типа `void *` можно присвоить переменной-указателю любого типа. Обычно функция `malloc` используется совместно с оператором `sizeof`. Например, следующая инструкция:

```
newPtr = malloc( sizeof( struct node ) );
```

определит размер в байтах структуры типа `struct node`, вычислив значение выражения `sizeof(struct node)`, *выделит новый блок памяти* данного размера и сохранит указатель на этот блок памяти в переменной `newPtr`. Выделяемая таким способом память *не* инициализируется. Если выделить память невозможно, `malloc` вернет `NULL`.

Функция `free` *освобождает* ранее выделенную память, то есть память возвращается системе, чтобы она могла быть вновь выделена в будущем. Чтобы *освободить* память, выделенную динамически предыдущим вызовом `malloc`, можно воспользоваться инструкцией

```
free( newPtr );
```

В языке C имеются также функции `calloc` и `realloc`, предназначенные для динамического выделения памяти и управления ею. Эти функции обсуждаются в разделе 14.9. В разделах, следующих ниже, обсуждаются структуры: списки, стеки, очереди и деревья, – каждая из которых создается за счет динамического выделения памяти и с применением структур, ссылающихся на самих себя.

Размер структуры не всегда является суммой размеров ее полей. Это обусловлено тем, что в разных аппаратных архитектурах предъявляются разные требования к выравниванию полей данных (см. главу 10).

Вызывая функцию `malloc`, проверяйте возвращаемый ею указатель на равенство значению `NULL`, которое указывает, что память не была выделена.

Отказ от возврата динамически выделенной памяти, когда она станет не нужна, может привести к исчерпанию памяти. Иногда эту проблему называют «утечкой памяти».

Когда динамически выделенная память станет не нужна программе, немедленно возвращайте ее системе с помощью функции `free`. После этого обязательно присваивайте переменной-указателю значение `NULL`, чтобы исключить возможность обращения к памяти после ее освобождения, которая к этому моменту может быть повторно выделена для других нужд.

Попытка освободить память, которая прежде не выделялась вызовом функции `malloc`*, является ошибкой.*

Попытки обратиться к памяти, которая уже была освобождена, часто приводят к аварийному завершению программы.

12.4 Связанные списки

Связанный список – это линейная коллекция структур, ссылающихся на самих себя, называемых **узлами**, связанных через поле **связи**, откуда и пошло название «связанный» список. Доступ к связанному списку осуществляется через указатель на его *первый* узел. Доступ к последующим узлам осуществляется через *поле связи*, имеющееся в каждом узле. В соответствии с соглашениями в поле связи последнего узла списка записывается значение `NULL`, служащее признаком конца списка. Данные сохраняются в связанном списке динамически – каждый новый узел создается по мере необходимости. Узел может хранить данные *любого* типа, в том числе и другие структуры. **Стеки** и **очереди** также являются линейными коллекциями структур и, как будет показано ниже, представляют ограниченные версии связанных списков. **Деревья** являются *нелинейными* коллекциями.

Списки данных можно хранить и в массивах, но связанные списки дают некоторые дополнительные преимущества. Связанные списки с успехом могут использоваться, когда количество элементов данных заранее *неизвестно*. Связанные списки имеют динамическую природу – длина списка может увеличиваться и уменьшаться *в процессе выполнения* по мере необходимости. В отличие от списков, размеры массивов определяются на этапе компиляции и не могут изменяться впоследствии. Массивы имеют ограниченный объем. Объем связанных списков ограничивается только доступной в системе памятью, выделенной для обслуживания динамических потребностей программы.

При объявлении массива можно указать его размер, превышающий ожидаемое количество элементов данных, но это может привести к напрасному расходованию памяти. Связанные списки обеспечивают более оптимальное использование памяти.

При работе со связанным списком можно поддерживать определенный порядок следования элементов, добавляя новые узлы в соответствующие места в списке.

Операции вставки элементов в отсортированный массив и удаления их из массива могут отнимать массу времени – все элементы, следующие за вставляемым или удаляемым, требуется сдвигать в ту или иную сторону.

Элементы массива хранятся в непрерывной области памяти. Это дает возможность непосредственного обращения к элементам массива, потому что адрес любого элемента легко вычисляется, исходя из смещения относительно начала массива. Связанные списки не обеспечивают такой возможности непосредственного обращения к их элементам.

Узлы связанного списка редко хранятся в непрерывной области памяти. Однако логически узлы связанного списка представляют собой непрерывную цепочку. На рис. 12.2 изображен связанный список с несколькими узлами.

Использование динамической памяти (вместо массивов) для хранения структур данных, способных увеличиваться и уменьшаться в размерах, может способствовать экономии памяти. Однако имейте в виду, что сами указатели также занимают место в памяти и что управление динамической памятью сопряжено с накладными расходами на вызовы функций.

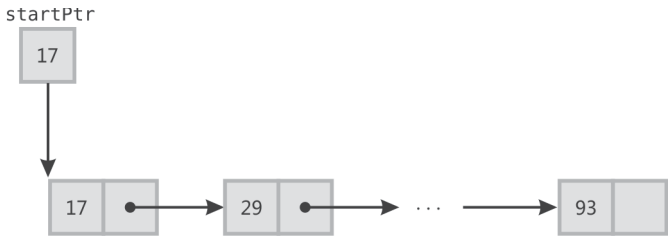


Рис. 12.2 | Графическое представление связанного списка

Программа в примере 12.1 (вывод программы представлен в примере 12.2) оперирует списком символов. С ее помощью можно вставлять символы в список в алфавитном порядке (функция `insert`) и удалять их из списка (функция `delete`). Подробное обсуждение программы следует ниже.

Пример 12.1 | Вставка узлов в список и удаление их из списка

```
 1 // Пример 12.1: fig12_03.c
 2 // Вставка узлов в список и удаление их из списка
 3 #include <stdio.h>
 4 #include <stdlib.h>
 5
 6 // структура, ссылающаяся на саму себя
 7 struct listNode {
 8    char data;                // каждая структура listNode содержит символ
 9    struct listNode *nextPtr; // указатель на следующий узел
10 }; // конец структуры listNode
11
```

```
12 typedef struct listNode ListNode; // синоним для struct listNode
13 typedef ListNode *ListNodePtr;    // синоним для ListNode*
14
15 // прототипы
16 void insert( ListNodePtr *sPtr, char value );
17 char delete( ListNodePtr *sPtr, char value );
18 int isEmpty( ListNodePtr sPtr );
19 void printList( ListNodePtr currentPtr );
20 void instructions( void );
21
22 int main( void )
23 {
24    ListNodePtr startPtr = NULL; // изначально список пуст
25    unsigned int choice;         // выбор пользователя
26    char item;                   // символ, введенный пользователем
27
28    instructions(); // вывести меню
29    printf( "%s", "? " );
30    scanf( "%u", &choice );
31
32    // выполнять цикл, пока пользователь не выберет пункт 3
33    while ( choice != 3 ) {
34
35       switch ( choice ) {
36          case 1:
37             printf( "%s", "Enter a character: " );
38             scanf( "\n%c", &item );
39             insert( &startPtr, item ); // вставить элемент в список
40             printList( startPtr );
41             break;
42          case 2: // удалить элемент
43             // если список не пуст
44             if ( !isEmpty( startPtr ) ) {
45                printf( "%s", "Enter character to be deleted: " );
46                scanf( "\n%c", &item );
47
48                // если символ найден, удалить его из списка
49                if ( delete( &startPtr, item ) ) { // удалить элемент
50                   printf( "%c deleted.\n", item );
51                   printList( startPtr );
52                } // конец if
53                else {
54                   printf( "%c not found.\n\n", item );
55                } // конец else
56             } // конец if
57             else {
58                puts( "List is empty.\n" );
59             } // конец else
60
61             break;
62          default:
63             puts( "Invalid choice.\n" );
64             instructions();
65             break;
66       } // конец switch
67
```

```
68       printf( "%s", "? " );
69       scanf( "%u", &choice );
70    } // конец while
71
72    puts( "End of run." );
73 } // конец main
74
75 // выводит инструкцию по работе с программой
76 void instructions( void )
77 {
78    puts( "Enter your choice:\n"
79       "   1 to insert an element into the list.\n"
80       "   2 to delete an element from the list.\n"
81       "   3 to end." );
82 } // конец функции instructions
83
84 // вставляет новое значение в список в порядке сортировки
85 void insert( ListNodePtr *sPtr, char value )
86 {
87    ListNodePtr newPtr;      // указатель на новый узел
88    ListNodePtr previousPtr; // указатель на предыдущий узел в списке
89    ListNodePtr currentPtr;  // указатель на текущий узел в списке
90
91    newPtr = malloc( sizeof( ListNode ) ); // создать узел
92
93    if ( newPtr != NULL ) {    // если память выделена
94       newPtr->data = value;   // записать значение в узел
95       newPtr->nextPtr = NULL; // узел пока не связан с другим узлом
96
97       previousPtr = NULL;
98       currentPtr = *sPtr;
99
100      // найти место в списке для вставки нового узла
101      while ( currentPtr != NULL && value > currentPtr->data ) {
102         previousPtr = currentPtr;        // перейти к ...
103         currentPtr = currentPtr->nextPtr; // ... следующему узлу
104      } // конец while
105
106      // вставить новый узел в начало списка
107      if ( previousPtr == NULL ) {
108         newPtr->nextPtr = *sPtr;
109         *sPtr = newPtr;
110      } // конец if
111      else { // вставить новый узел между previousPtr и currentPtr
112         previousPtr->nextPtr = newPtr;
113         newPtr->nextPtr = currentPtr;
114      } // конец else
115   } // конец if
116   else {
117      printf( "%c not inserted. No memory available.\n", value );
118   } // конец else
119 } // конец функции insert
120
121 // удаляет элемент из списка
122 char delete( ListNodePtr *sPtr, char value )
123 {
124    ListNodePtr previousPtr; // указатель на предыдущий узел в списке
```

```
125    ListNodePtr currentPtr;  // указатель на текущий узел в списке
126    ListNodePtr tempPtr;     // временный указатель на узел
127
128    // удалить первый узел
129    if ( value == ( *sPtr )->data ) {
130       tempPtr = *sPtr; // сохранить указатель на удаляемый узел
131       *sPtr = ( *sPtr )->nextPtr; // исключить узел из списка
132       free( tempPtr ); // освободить память, занимаемую исключенным узлом
133       return value;
134    } // конец if
135    else {
136       previousPtr = *sPtr;
137       currentPtr = ( *sPtr )->nextPtr;
138
139       // найти элемент списка с указанным символом
140       while ( currentPtr != NULL && currentPtr->data != value ) {
141          previousPtr = currentPtr;         // перейти к ...
142          currentPtr = currentPtr->nextPtr; // ... следующему узлу
143       } // конец while
144
145       // удалить узел currentPtr
146       if ( currentPtr != NULL ) {
147          tempPtr = currentPtr;
148          previousPtr->nextPtr = currentPtr->nextPtr;
149          free( tempPtr );
150          return value;
151       } // конец if
152    } // конец else
153
154    return '\0';
155 } // конец функции delete
156
157 // возвращает 1, если список пуст, 0 - в противном случае
158 int isEmpty( ListNodePtr sPtr )
159 {
160    return sPtr == NULL;
161 } // конец функции isEmpty
162
163 // выводит список
164 void printList( ListNodePtr currentPtr )
165 {
166    // если список пуст
167    if ( isEmpty( currentPtr ) ) {
168       puts( "List is empty.\n" );
169    } // конец if
170    else {
171       puts( "The list is:" );
172
173       // пока не достигнут конец списка
174       while ( currentPtr != NULL ) {
175          printf( "%c --> ", currentPtr->data );
176          currentPtr = currentPtr->nextPtr;
177       } // конец while
178
179       puts( "NULL\n" );
180    } // конец else
181 } // конец функции printList
```

Пример 12.2 | Вывод программы из примера 12.1

```
Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 1
Enter a character: B
The list is:
B --> NULL

? 1
Enter a character: A

The list is:
A --> B --> NULL

? 1
Enter a character: C
The list is:
A --> B --> C --> NULL

? 2
Enter character to be deleted: D
D not found.

? 2
Enter character to be deleted: B
B deleted.
The list is:
A --> C --> NULL

? 2
Enter character to be deleted: C
C deleted.
The list is:
A --> NULL

? 2
Enter character to be deleted: A
A deleted.
List is empty.

? 4
Invalid choice.

Enter your choice:
   1 to insert an element into the list.
   2 to delete an element from the list.
   3 to end.
? 3
End of run.
```

Основными функциями для работы со связанными списками являются `insert` (строки 85–119) и `delete` (строки 122–155). Функция `isEmpty` (строки

158–161) называется *функцией-предикатом* – она *не* изменяет список, а просто определяет, является ли список пустым (то есть является ли пустым указатель на первый элемент списка). Если список пуст, возвращается `1`; в противном случае возвращается `0`. [*Обратите внимание*: если используется компилятор, совместимый со стандартом C11, вы можете воспользоваться типом `_Bool` (раздел 4.10) вместо `int`.] Функция `printList` (строки 164–181) выводит содержимое списка.

12.4.1 Функция insert

Символы вставляются в список в *алфавитном порядке*. Функция `insert` (строки 85–119) принимает адрес списка и символ для вставки. Адрес списка необходим, чтобы иметь возможность вставить значение *в начало* списка. Передача адреса списка (адреса указателя на первый узел в списке), то есть по ссылке, обеспечивает возможность его *изменения*. Так как сам список является указателем (на первый элемент), при передаче адреса указателя создается **указатель на указатель** (этакая **двойная косвенность**). Это довольно сложный прием, требующий очень внимательного отношения к программированию. При вставке символа в список выполняются следующие действия:

1. Вызовом функции `malloc` *создается новый узел*, адрес которого присваивается переменной `newPtr` (строка 91), в поле `newPtr->data` сохраняется символ, предназначенный для вставки (строка 94), а полю `newPtr->nextPtr` присваивается значение `NULL` (строка 95).
2. Указатель `previousPtr` инициализируется значением `NULL` (строка 97), а указатель `currentPtr` – значением `*sPtr` (строка 98) – указателем на начало списка. Указатели `previousPtr` и `currentPtr` хранят адреса узлов, *предшествующего* новому узлу и *следующего* за новым узлом.
3. Пока указатель `currentPtr` не равен `NULL` и значение для вставки больше значения поля `currentPtr->data` (строка 101), значение `currentPtr` присваивается указателю `previousPtr` (строка 102) и затем `currentPtr` перемещается на следующий узел в списке (строка 103). Таким способом определяется *позиция для вставки* нового узла в списке.
4. Если `previousPtr` оказался равным `NULL` (строка 107), следовательно, новый узел должен быть вставлен как *первый* узел списка (строки 108–109). Значение `*sPtr` присваивается полю `newPtr->nextPtr` (*новый узел* связывается с *прежним первым элементом* списка), а значение `newPtr` присваивается указателю `*sPtr` (теперь `*sPtr` указывает на *новый узел*). Иначе, если указатель `previousPtr` не равен `NULL`, новый узел вставляется в тело списка (строки 112–113). Указатель `newPtr` присваивается полю `previousPtr->nextPtr` (*предыдущий* узел связы-

вается с *новым* узлом), и указатель `currentPtr` присваивается полю `newPtr->nextPtr` (*новый* узел связывается с *текущим* узлом).

Всегда присваивайте значение `NULL` *полю связи вновь созданного узла. Указатели должны инициализироваться перед их использованием.*

На рис. 12.3 изображен процесс вставки узла с символом `'C'` в упорядоченный список. В верхней половине (а) на рисунке изображены список и новый узел непосредственно перед вставкой. В нижней половине (б) изображен результат вставки нового узла. Переназначенные указатели изображены пунктирными стрелками. Для простоты мы реализовали функцию `insert` (и другие родственные функции в этой главе) как не имеющую возвращаемого значения. Однако, поскольку есть вероятность, что вызов `malloc` потерпит неудачу, было бы неплохо, если бы функция `insert` возвращала признак успешности операции.

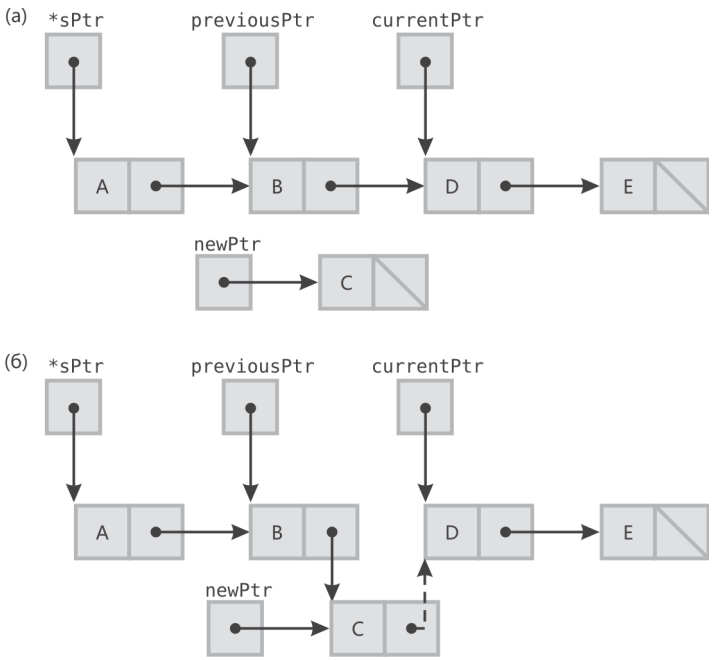


Рис. 12.3 | Вставка узла в упорядоченный список

12.4.2 Функция delete

Функция `delete` (строки 122–155 в примере 12.1) принимает адрес указателя на начало списка и удаляемый символ. При вставке символа в список выполняются следующие действия (см. рис. 12.4):

1. Если удаляемый символ совпадает с символом в *первом* узле (строка 129), значение `*sPtr` присваивается указателю `tempPtr` (`tempPtr` передается функции `free` для освобождения памяти, ставшей ненужной), значение поля `(*sPtr)->nextPtr` присваивается указателю `*sPtr` (теперь `*sPtr` указывает на *второй* узел в списке), блок памяти, на который указывает `tempPtr`, освобождается вызовом функции `free`, и удаленный символ возвращается вызывающей программе.
2. В противном случае указатель `previousPtr` инициализируется значением `*sPtr`, а указатель `currentPtr` – значением `(*sPtr)->nextPtr` (строки 136–137), чтобы перейти ко второму узлу.
3. Пока `currentPtr` не равен `NULL` и удаляемое значение не равно значению `currentPtr->data` (строка 140), значение `currentPtr` присваивается указателю `previousPtr` (строка 141) и значение поля `currentPtr ->nextPtr` – указателю `currentPtr` (строка 142). Таким способом выполняется поиск узла, подлежащего удалению, если он вообще присутствует в списке.
4. Если `currentPtr` не равен `NULL` (строка 146), значение `currentPtr` присваивается указателю `tempPtr` (строка 147), значение `currentPtr`

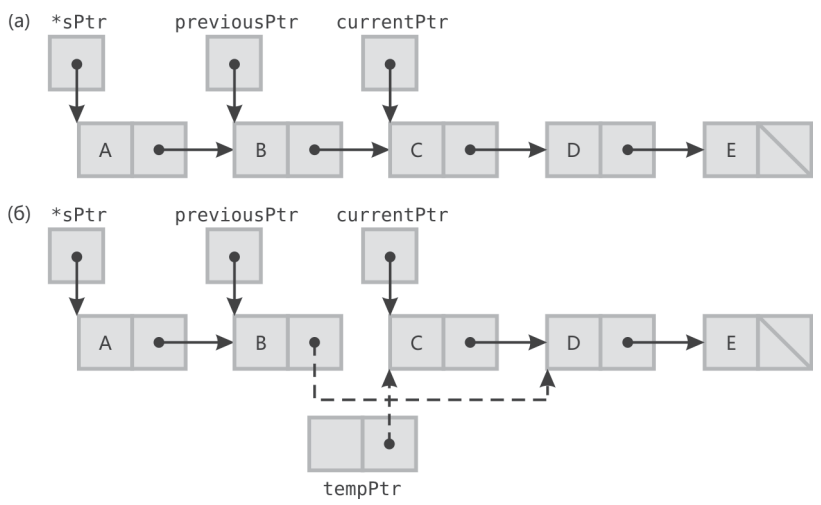


Рис. 12.4 | Удаление узла из списка

->nextPtr – полю previousPtr->nextPtr (строка 148), блок памяти, на который указывает tempPtr (строка 149), освобождается вызовом функции free, и удаленный символ возвращается вызывающей программе (строка 150). Если currentPtr равен NULL, программе возвращается нулевой символ ('\0'), чтобы сообщить ей, что требуемый символ не был найден в списке (строка 154).

На рис. 12.4 изображен процесс удаления узла из списка. В верхней половине (а) на рисунке изображен список, полученный после предыдущей операции вставки. В нижней половине (б) изображено, как выполняется переназначение поля связи в previousPtr и присваивание значения currentPtr указателю tempPtr. Указатель tempPtr используется для освобождения памяти, выделенной для узла, хранящего символ 'C'. Обратите внимание, что в строках 132 и 149 (в примере 12.1) мы передаем функции free указатель tempPtr. Напомню, что после освобождения памяти соответствующим указателям рекомендуется присваивать значение NULL. Мы не делаем этого в данных случаях потому, что tempPtr является локальной автоматической переменной и функция, где эта переменная объявлена, возвращает управление сразу после освобождения памяти.

12.4.3 Функция printList

Функция printList (строки 164–181) принимает указатель currentPtr на начало списка. Сначала функция проверяет, не является ли список пустым (строки 167–169), и если он пуст, выводит строку "List is empty." (Список пуст) и завершается. В противном случае она выводит данные, хранящиеся в списке (строки 170–180). Пока currentPtr не равен NULL, значение поля currentPtr->data выводится функцией, и затем значение currentPtr->nextPtr присваивается указателю currentPtr, чтобы перейти к следующему узлу. Если поле связи последнего узла в списке не будет равно NULL, алгоритм вывода попытается *выйти за пределы списка* и тем самым вызовет ошибку. Алгоритм вывода одинаков как для списков, так и для очередей и стеков.

12.5 Стеки

Стек может быть реализован как ограниченная версия связанного списка. Новые узлы могут добавляться в стек и удаляться из стека *только* с одного конца – с его *вершины*. По этой причине стек также называют структурой данных типа «**последним пришел, первым вышел**» (Last-In, First-out, LIFO). Доступ к стеку осуществляется посредством указателя на его вершину. Поле связи последнего узла стека устанавливается в значение NULL, чтобы обозначить дно стека.

На рис. 12.5 изображен стек с несколькими узлами – переменная stackPtr указывает на вершину стека. Стеки и связанные списки имеют совершенно

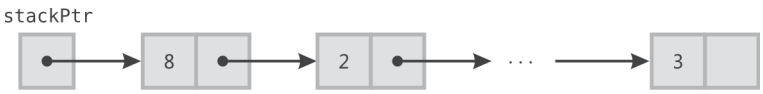


Рис. 12.5 | Графическое представление стека

идентичную организацию. Разница лишь в том, что операции вставки и удаления в связанных списках могут выполняться над любыми элементами, а в стеках – только с элементом на *вершине* стека.

Если забыть присвоить полю связи последней структуры в стеке значение `NULL`*, это может привести к ошибке во время выполнения.*

Основными функциями, предназначенными для работы со стеками, являются функции `push` и `pop`. Функция `push` создает новый узел и помещает его на *вершину* стека. Функция `pop` *удаляет* узел с *вершины* стека, *освобождает* занимаемую им память и *возвращает значение узла*.

Программа в примере 12.3 (вывод программы представлен в примере 12.4) реализует простой стек для хранения целых чисел. Она предлагает пользователю на выбор три операции: 1) втолкнуть значение в стек (функция `push`), 2) вытолкнуть значение из стека (функция `pop`) и 3) завершить программу.

Пример 12.3 | Реализация простого стека

```
// Пример 12.3: fig12_08.c
// Реализация простого стека
#include <stdio.h>
#include <stdlib.h>

// Структура, ссылающаяся на саму себя
struct stackNode {
   int data;                  // определить данные с типом int
   struct stackNode *nextPtr; // указатель на структуру stackNode
}; // конец структуры stackNode

typedef struct stackNode StackNode; // синоним для struct stackNode
typedef StackNode *StackNodePtr;    // синоним для StackNode*

// прототипы
void push( StackNodePtr *topPtr, int info );
int pop( StackNodePtr *topPtr );
int isEmpty( StackNodePtr topPtr );
void printStack( StackNodePtr currentPtr );
void instructions( void );

// выполнение программы начинается с функции main
int main( void )
{
```

```
25    StackNodePtr stackPtr = NULL; // указатель на вершину стека
26    unsigned int choice;          // пункт меню, выбранный пользователем
27    int value;                    // целое число, введенное пользователем
28
29    instructions(); // вывести меню
30    printf( "%s", "? " );
31    scanf( "%u", &choice );
32
33    // пока пользователь не выберет пункт меню 3
34    while ( choice != 3 ) {
35
36       switch ( choice ) {
37          // втолкнуть значение в стек
38          case 1:
39             printf( "%s", "Enter an integer: " );
40             scanf( "%d", &value );
41             push( &stackPtr, value );
42             printStack( stackPtr );
43             break;
44          // вытолкнуть значение со стека
45          case 2:
46             // если стек не пуст
47             if ( !isEmpty( stackPtr ) ) {
48                printf( "The popped value is %d.\n", pop( &stackPtr ) );
49             } // конец if
50
51             printStack( stackPtr );
52             break;
53          default:
54             puts( "Invalid choice.\n" );
55             instructions();
56             break;
57       } // конец switch
58
59       printf( "%s", "? " );
60       scanf( "%u", &choice );
61    } // конец while
62
63    puts( "End of run." );
64 } // конец main
65
66 // выводит инструкцию по использованию программы
67 void instructions( void )
68 {
69    puts( "Enter choice:\n"
70       "1 to push a value on the stack\n"
71       "2 to pop a value off the stack\n"
72       "3 to end program" );
73 } // конец функции instructions
74
75 // вставляет узел на вершину стека
76 void push( StackNodePtr *topPtr, int info )
77 {
```

```
   StackNodePtr newPtr; // указатель на новый узел

   newPtr = malloc( sizeof( StackNode ) );

   // вставить узел на вершину стека
   if ( newPtr != NULL ) {
      newPtr->data = info;
      newPtr->nextPtr = *topPtr;
      *topPtr = newPtr;
   } // конец if
   else { // отсутствует место в памяти
      printf( "%d not inserted. No memory available.\n", info );
   } // конец else
} // конец функции push

// удаляет узел с вершины стека
int pop( StackNodePtr *topPtr )
{
   StackNodePtr tempPtr; // временный указатель на узел
   int popValue;         // значение узла

   tempPtr = *topPtr;
   popValue = ( *topPtr )->data;
   *topPtr = ( *topPtr )->nextPtr;
   free( tempPtr );
    return popValue;
} // конец функции pop

// выводит содержимое стека
void printStack( StackNodePtr currentPtr )
{
   // если стек пуст
   if ( currentPtr == NULL ) {
      puts( "The stack is empty.\n" );
   } // конец if
   else {
      puts( "The stack is:" );

      // пока не достигнут конец стека
      while ( currentPtr != NULL ) {
         printf( "%d --> ", currentPtr->data );
         currentPtr = currentPtr->nextPtr;
      } // конец while

      puts( "NULL\n" );
   } // конец else
} // конец функции printList

// возвращает 1, если стек пуст, 0 - в противном случае
int isEmpty( StackNodePtr topPtr )
{
   return topPtr == NULL;
} // конец функции isEmpty
```

Пример 12.4 | Вывод программы из примера 12.3

```
Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 1
Enter an integer: 5
The stack is:
5 --> NULL

? 1
Enter an integer: 6
The stack is:
6 --> 5 --> NULL

? 1
Enter an integer: 4
The stack is:
4 --> 6 --> 5 --> NULL

? 2
The popped value is 4.
The stack is:
6 --> 5 --> NULL
? 2
The popped value is 6.
The stack is:
5 --> NULL

? 2
The popped value is 5.
The stack is empty.

? 2
The stack is empty.

? 4
Invalid choice.

Enter choice:
1 to push a value on the stack
2 to pop a value off the stack
3 to end program
? 3
End of run.
```

12.5.1 Функция push

Функция `push` (строки 76–91) помещает новый узел на вершину стека, выполняя эту операцию в три этапа:

1. Создает *новый узел* вызовом `malloc` и сохраняет адрес выделенного блока памяти в указателе `newPtr` (строка 80).

2. Присваивает полю `newPtr->data` значение, которое требуется поместить в стек (строка 84), а в *поле связи* `newPtr->nextPtr` сохраняет значение `*topPtr` (*указатель на вершину стека*) (строка 85), в результате *поле связи* в `newPtr` теперь ссылается на узел, *находящийся* на вершине стека.
3. Сохраняет `newPtr` в `*topPtr` (строка 86) – теперь `*topPtr` указывает на *новую* вершину стека.

Операции с `*topPtr` изменяют значение `stackPtr` в функции `main`. На рис. 12.6 изображен порядок действий, выполняемых функцией `push`. В верхней половине рисунка (а) изображены стек и новый узел *перед* выполнением операции `push`. В нижней половине рисунка (б) – *этапы 2 и 3*, которые превращают узел со значением `12` в новую вершину стека.

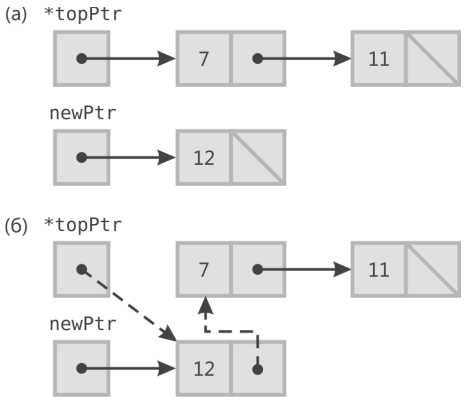


Рис. 12.6 | Операция push

12.5.2 Функция pop

Функция `pop` (строки 94–104 в примере 12.4) удаляет узел с вершины стека. Перед вызовом `pop` функция `main` проверяет наличие хотя бы одного узла в стеке. Операция `pop` выполняется в пять этапов:

1. Значение `*topPtr` присваивается переменной `tempPtr` (строка 99); указатель `tempPtr` будет использоваться *для освобождения* памяти.
2. Значение `(*topPtr)->data`, находящееся на вершине стека, *сохраняется* в переменной `popValue` (строка 100).
3. Значение поля связи `(*topPtr)->nextPtr` сохраняется по адресу `*topPtr` (строка 101), чтобы передвинуть указатель `*topPtr` *на новую вершину стека*.

4. Память, на которую указывает `tempPtr` (строка 102), освобождается.
5. Вызывающей программе возвращается значение `popValue` (строка 103).

На рис. 12.7 изображен порядок действий, выполняемых функцией `pop`. В верхней половине рисунка (а) изображен стек, получившийся сразу *после* выполнения предыдущей операции `push`. В нижней половине рисунка (б) изображены указатель `tempPtr`, ссылающийся на *первый* узел стека, и указатель `topPtr`, ссылающийся на второй узел стека. Функция `free` используется для *освобождения памяти*, на которую ссылается указатель `tempPtr`.

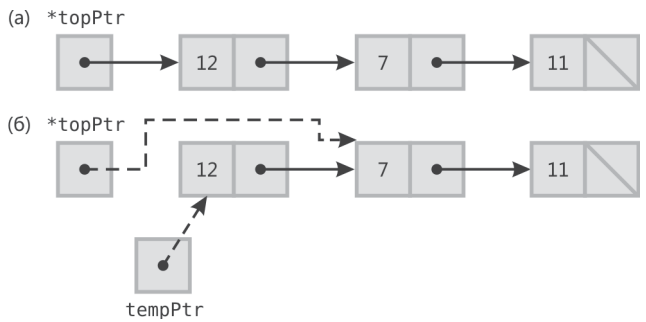


Рис. 12.7 | Операция push

12.5.3 Области применения стеков

Стеки имеют массу интереснейших применений. Например, когда выполняется *вызов функции*, функция должна знать, как *вернуть управление* вызывающей программе, с этой целью *адрес возврата* помещается в стек. Если выполняется целый каскад вызовов функций, на стек последовательно помещаются адреса возвратов в порядке «последним пришел, первым вышел» (Last-In, First-out, LIFO), чтобы каждая вызванная функция могла вернуть управление в точку ее вызова. Стеки поддерживают также рекурсивные вызовы функций, обслуживая их как обычные вызовы функций.

Помимо этого, на стеке выделяется дополнительное пространство для хранения автоматических переменных, объявленных в вызываемых функциях. Когда функция возвращает управление, зарезервированное пространство для автоматических переменных выталкивается со стека, и они становятся недоступными для программы. Стеки используются компиляторами в процессе вычисления выражений и создания машинного кода.

12.6 Очереди

Еще одной распространенной структурой данных являются **очереди**. Очередь представляет собой аналог очереди, например, в магазине – *первый* человек в такой очереди *обслуживается первым*, остальные клиенты становятся в *конец* очереди и обслуживаются *в порядке поступления*. Узлы удаляются *только* **из головы очереди**, а добавляются только в конец, или в **хвост очереди**. Из-за этой особенности очереди называют структурами данных, обслуживаемыми по принципу «первым пришел, первым вышел» (First-In, First-out, FIFO). Операции вставки узла в очередь и удаления его из очереди называют `enqueue` и `dequeue` соответственно.

Очереди имеют множество практических применений в компьютерных системах. Компьютеры, имеющие единственный процессор, в каждый конкретный момент времени способны обслуживать только одного пользователя. Записи с информацией о пользователях помещаются в очередь, и они постепенно продвигаются к началу очереди *по мере обслуживания*. Запись, находящаяся *в голове* очереди, *обслуживается первой*.

Очереди также применяются для поддержки *очередей печати*. В многопользовательской системе может иметься только один принтер. Многие пользователи могут отправлять свои документы на печать. В то время пока принтер занят, могут готовиться другие задания для печати. Все они помещаются в *очередь*, где ожидают освобождения принтера.

В компьютерных сетях пакеты также помещаются в очереди. Каждый раз, когда пакет поступает в сетевой узел, выполняющий маршрутизацию, его необходимо передать дальше по сети, согласно адресу назначения. Маршрутизация пакетов выполняется по одному за раз, поэтому вновь поступающие пакеты помещаются в очередь, где ожидают, пока маршрутизатор обслужит их. На рис. 12.8 изображена очередь с несколькими узлами. Обратите внимание, что очередь адресуется двумя указателями, ссылающимися на голову и хвост очереди.

Программа в примере 12.5 (вывод программы представлен в примере 12.6) реализует простую очередь. Она предлагает на выбор три операции:

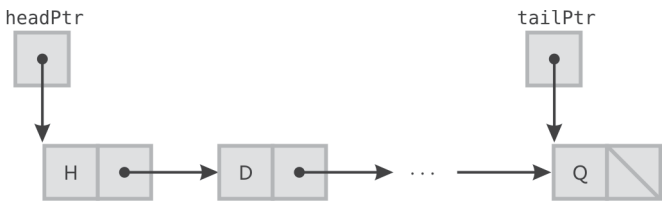


Рис. 12.8 | Графическое представление очереди

вставку узла в очередь (функция enqueue), *удаление* узла из очереди (функция dequeue) и завершение программы.

Пример 12.5 | Реализация простой очереди

```
// Пример 12.5: fig12_13.c
// Реализация простой очереди
#include <stdio.h>
#include <stdlib.h>

// структура, ссылающаяся на саму себя
struct queueNode {
   char data;                 // определить данные с типом char
   struct queueNode *nextPtr; // указатель на структуру queueNode
}; // конец структуры queueNode

typedef struct queueNode QueueNode;
typedef QueueNode *QueueNodePtr;

// прототипы функций
void printQueue( QueueNodePtr currentPtr );
int isEmpty( QueueNodePtr headPtr );
char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr );
void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
              char value );
void instructions( void );

// выполнение программы начинается с функции main
int main( void )
{
   QueueNodePtr headPtr = NULL; // инициализировать headPtr
   QueueNodePtr tailPtr = NULL; // инициализировать tailPtr
   unsigned int choice;         // пункт меню, выбранный пользователем
   char item;                   // символ, введенный пользователем

   instructions();              // вывести меню
   printf( "%s", "? " );
   scanf( "%u", &choice );

   // пока пользователь не выберет пункт меню 3
   while ( choice != 3 ) {

      switch( choice ) {
         // поставить значение в очередь
         case 1:
            printf( "%s", "Enter a character: " );
            scanf( "\n%c", &item );
            enqueue( &headPtr, &tailPtr, item );
            printQueue( headPtr );
            break;
         // удалить значение из очереди
```

```
47          case 2:
48             // если очередь не пуста
49             if ( !isEmpty( headPtr ) ) {
50                item = dequeue( &headPtr, &tailPtr );
51                printf( "%c has been dequeued.\n", item );
52             } // конец if
53
54             printQueue( headPtr );
55             break;
56          default:
57             puts( "Invalid choice.\n" );
58             instructions();
59             break;
60       } // конец switch
61
62       printf( "%s", "? " );
63       scanf( "%u", &choice );
64    } // конец while
65
66    puts( "End of run." );
67 } // конец main
68
69 // display program instructions to user
70 void instructions( void )
71 {
72    printf ( "Enter your choice:\n"
73             "   1 to add an item to the queue\n"
74             "   2 to remove an item from the queue\n"
75             "   3 to end\n" );
76 } // конец функции instructions
77
78 // вставляет узел в хвост очереди
79 void enqueue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr,
80               char value )
81 {
82    QueueNodePtr newPtr; // указатель на новый узел
83
84    newPtr = malloc( sizeof( QueueNode ) );
85
86    if ( newPtr != NULL ) { // память доступна
87       newPtr->data = value;
88       newPtr->nextPtr = NULL;
89
90       // если очередь пуста, вставить узел в голову очереди
91       if ( isEmpty( *headPtr ) ) {
92          *headPtr = newPtr;
93       } // конец if
94       else {
95          ( *tailPtr )->nextPtr = newPtr;
96       } // конец else
97
```

```
 98       *tailPtr = newPtr;
 99    } // конец if
100    else {
101       printf( "%c not inserted. No memory available.\n", value );
102    } // конец else
103 } // конец функции enqueue
104
105 // удаляет узел из головы очереди
106 char dequeue( QueueNodePtr *headPtr, QueueNodePtr *tailPtr )
107 {
108    char value;           // значение узла
109    QueueNodePtr tempPtr; // временный указатель на узел
110
111    value = ( *headPtr )->data;
112    tempPtr = *headPtr;
113    *headPtr = ( *headPtr )->nextPtr;
114
115    // если очередь пуста
116    if ( *headPtr == NULL ) {
117       *tailPtr = NULL;
118    } // конец if
119
120    free( tempPtr );
121    return value;
122 } // конец функции dequeue
123
124 // возвращает 1, если очередь пуста, 0 — в противном случае
125 int isEmpty( QueueNodePtr headPtr )
126 {
127    return headPtr == NULL;
128 } // конец функции isEmpty
129
130 // выводит содержимое очереди
131 void printQueue( QueueNodePtr currentPtr )
132 {
133    // если очередь пуста
134    if ( currentPtr == NULL ) {
135       puts( "Queue is empty.\n" );
136    } // конец if
137    else {
138       puts( "The queue is:" );
139
140       // пока не достигнут конец очереди
141       while ( currentPtr != NULL ) {
142          printf( "%c --> ", currentPtr->data );
143          currentPtr = currentPtr->nextPtr;
144       } // конец while
145
146       puts( "NULL\n" );
147    } // конец else
148 } // конец функции printQueue
```

Пример 12.6 | Вывод программы из примера 12.5

```
Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end

? 1
Enter a character: A
The queue is:
A --> NULL

? 1
Enter a character: B
The queue is:
A --> B --> NULL

? 1
Enter a character: C
The queue is:
A --> B --> C --> NULL

? 2
A has been dequeued.
The queue is:
B --> C --> NULL

? 2
B has been dequeued.
The queue is:
C --> NULL

? 2
C has been dequeued.
Queue is empty.

? 2
Queue is empty.
? 4
Invalid choice.

Enter your choice:
   1 to add an item to the queue
   2 to remove an item from the queue
   3 to end
? 3
End of run.
```

12.6.1 Функция enqueue

Функция `enqueue` (строки 79–103) принимает три аргумента: *адрес указателя на голову очереди*, *адрес указателя на хвост очереди* и *значение*, добавляемое в очередь. Она выполняется в три этапа:

1. Создает новый узел: вызывает `malloc`, присваивает адрес выделенного блока памяти переменной `newPtr` (строка 84), присваивает значение для вставки в очередь полю `newPtr->data` (строка 87) и присваивает `NULL` полю связи `newPtr->nextPtr` (строка 88).
2. Если очередь пуста (строка 91), значение `newPtr` сохраняется в `*headPtr` (строка 92), потому что новый узел будет одновременно и головой, и хвостом очереди; в противном случае значение `newPtr` сохраняется в поле связи `(*tailPtr)->nextPtr` (строка 95), потому что новый узел должен быть помещен в очередь после узла в хвосте очереди.
3. Сохраняет `newPtr` в `*tailPtr` (строка 98), потому что новый узел всегда является хвостом очереди.

На рис. 12.9 изображен порядок действий, выполняемых функцией `enqueue`. В верхней половине рисунка (а) изображены очередь и новый узел *перед* выполнением операции. Пунктирные стрелки в нижней части рисунка (б) соответствуют *этапам 2 и 3*, в процессе выполнения которых новый узел добавляется в *конец* непустой очереди.

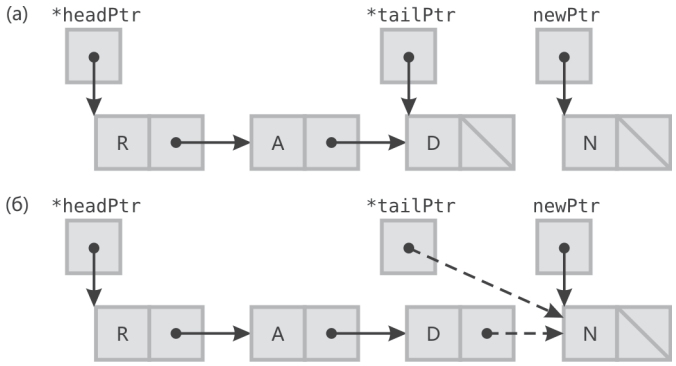


Рис. 12.9 | Операция enqueue

12.6.2 Функция dequeue

Функция (строки 106–122) принимает *адрес указателя головы очереди* и *адрес указателя хвоста очереди* и удаляет *первый* узел. Операция `dequeue` выполняется в шесть этапов:

1. Переменной `value` присваивается значение поля `(*headPtr)->data`, чтобы сохранить данные (строка 111).
2. Значение `*headPtr` присваивается указателю `tempPtr` (строка 112), который будет использоваться для освобождения памяти.

3. Значение `(*headPtr)->nextPtr` сохраняется по адресу `*headPtr` (строка 113) – теперь `*headPtr` указывает на новый первый узел в очереди.
4. Если `*headPtr` содержит значение `NULL` (строка 116), это же значение сохраняется по адресу `*tailPtr` (строка 117), так как подобная ситуация соответствует пустой очереди.
5. Освобождается блок памяти, на который указывает `tempPtr` (строка 120).
6. Вызывающей программе возвращается значение `value` (строка 121).

На рис. 12.10 изображен порядок действий, выполняемых функцией `dequeue`. В верхней половине рисунка (а) изображена очередь, получившаяся *после* выполнения предыдущей операции `enqueue`. В нижней половине рисунка (б) изображены указатель `tempPtr`, ссылающийся на узел, *извлеченный из очереди*, и указатель `headPtr`, ссылающийся на *новый первый узел* в очереди. Функция `free` используется для *освобождения памяти*, на которую ссылается указатель `tempPtr`.

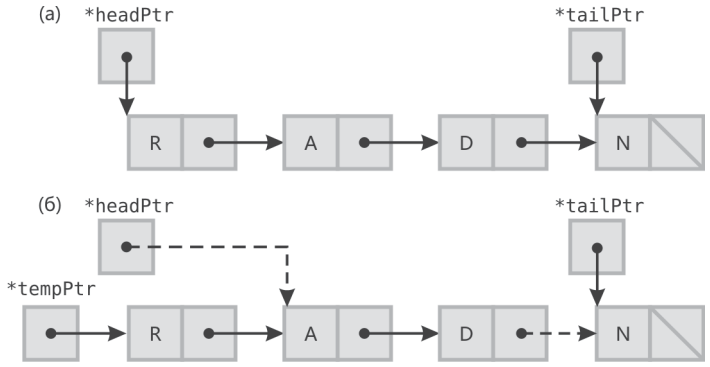


Рис. 12.10 | Операция dequeue

12.7 Деревья

Связанные списки, стеки и очереди – все эти структуры данных являются **линейными**. **Дерево** – это *нелинейная двумерная структура данных* с особыми свойствами. Узлы деревьев содержат *два или более* полей связи. В этом разделе мы познакомимся с **двоичными деревьями** (рис. 12.11) – деревьями, узлы которых содержат по *два* поля связи (каждое из которых, оба или ни одно из них может иметь значение `NULL`). Первый узел дерева называют **корневым узлом**. Каждое поле связи в узле дерева ссылается на **дочерний узел**. **Левый дочерний узел** – это *первый* узел **левого поддерева**, а **правый дочерний**

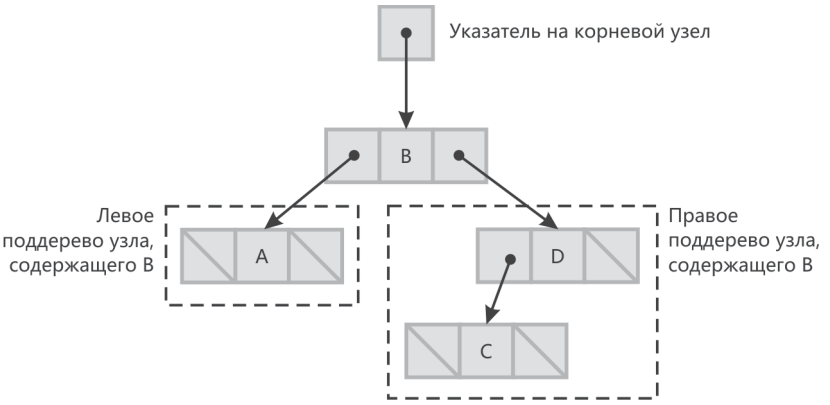


Рис. 12.11 | Графическое представление двоичного дерева

узел – *первый* узел **правого поддерева**. Дочерние узлы данного узла дерева между собой являются **братскими**. Узел, *не* имеющий дочерних узлов, называется **листом**. Обычно деревья изображаются в *перевернутом* виде – корневой узел вверху, а остальные узлы внизу.

В этом разделе мы создадим специальную разновидность двоичного дерева – **дерево двоичного поиска**. Дерево двоичного поиска (отличающееся от обычного дерева отсутствием узлов с повторяющимися значениями) имеет важную характеристику – значение любого левого поддерева меньше значения своего родительского узла, а значение любого правого поддерева больше значения своего **родительского узла**. На рис. 12.12 изображено дерево двоичного поиска с 12 значениями. Форма дерева двоичного поиска, соответствующего некоторому набору данных, может *отличаться* в зависимости от *порядка сортировки* значений в дереве.

Если забыть присвоить полям связи узлов-листьев дерева значение `NULL`*, это может привести к ошибке во время выполнения.*

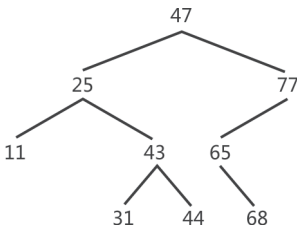


Рис. 12.12 | Дерево двоичного поиска

Программа в примере 12.7 (вывод программы представлен в примере 12.8) создает дерево двоичного поиска и выполняет обход его узлов тремя способами – **симметричный обход** (inorder), **обход в прямом направлении** (preorder) и **обход в обратном направлении** (postorder). Программа генерирует 10 случайных чисел и вставляет их в дерево, исключение составляют повторяющиеся значения, которые просто отбрасываются.

Пример 12.7 | Создание и обход дерева двоичного поиска

```
// Пример 12.7: fig12_19.c
// Создание и обход дерева двоичного поиска

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// структура, ссылающаяся на саму себя
struct treeNode {
   struct treeNode *leftPtr;  // указатель не левое поддерево
   int data;                  // значение узла
   struct treeNode *rightPtr; // указатель на правое поддерево
}; // конец структуры treeNode

typedef struct treeNode TreeNode; // синоним для типа struct treeNode
typedef TreeNode *TreeNodePtr;    // синоним для TreeNode*

// прототипы
void insertNode( TreeNodePtr *treePtr, int value );
void inOrder( TreeNodePtr treePtr );
void preOrder( TreeNodePtr treePtr );
void postOrder( TreeNodePtr treePtr );

// выполнение программы начинается с функции main
int main( void )
{
   unsigned int i; // счетчик для цикла (1-10)
   int item;       // переменная для хранения случайных значений
   TreeNodePtr rootPtr = NULL; // изначельно дерево не имеет узлов

   srand( time( NULL ) );
   puts( "The numbers being placed in the tree are:" );

   // вставить случайные значения от 0 до 14 в дерево
   for ( i = 1; i <= 10; ++i ) {
      item = rand() % 15;
      printf( "%3d", item );
      insertNode( &rootPtr, item );
   } // конец for

   // обойти дерево в прямом порядке
   puts( "\n\nThe preOrder traversal is:" );
   preOrder( rootPtr );

   // выполнить симметричный обход дерева
   puts( "\n\nThe inOrder traversal is:" );
```

```
47    inOrder( rootPtr );
48
49    // обойти дерево в обратном порядке
50    puts( "\n\nThe postOrder traversal is:" );
51    postOrder( rootPtr );
52 } // конец main
53
54 // вставляет узел в дерево
55 void insertNode( TreeNodePtr *treePtr, int value )
56 {
57    // если дерево пустое
58    if ( *treePtr == NULL ) {
59       *treePtr = malloc( sizeof( TreeNode ) );
60
61       // если память успешно выделена - сохранить данные
62       if ( *treePtr != NULL ) {
63          ( *treePtr )->data = value;
64          ( *treePtr )->leftPtr = NULL;
65          ( *treePtr )->rightPtr = NULL;
66       } // конец if
67       else {
68          printf( "%d not inserted. No memory available.\n", value );
69       } // конец else
70    } // конец if
71    else { // дерево не пустое
72       // значение для вставки меньше значения текущего узла
73       if ( value < ( *treePtr )->data ) {
74          insertNode( &( ( *treePtr )->leftPtr ), value );
75       } // конец if
76
77       // значение для вставки больше значения текущего узла
78       else if ( value > ( *treePtr )->data ) {
79          insertNode( &( ( *treePtr )->rightPtr ), value );
80       } // конец else if
81       else { // игнорировать повторяющиеся значения
82          printf( "%s", "dup" );
83       } // конец else
84    } // конец else
85 } // конец функции insertNode
86
87 // выполняет симметричный обход дерева
88 void inOrder( TreeNodePtr treePtr )
89 {
90    // если дерево не пустое, выполнить обход
91    if ( treePtr != NULL ) {
92       inOrder( treePtr->leftPtr );
93       printf( "%3d", treePtr->data );
94       inOrder( treePtr->rightPtr );
95    } // конец if
96 } // конец функции inOrder
97
98 // выполняет обход дерева в прямом порядке
99 void preOrder( TreeNodePtr treePtr )
100 {
101    // если дерево не пустое, выполнить обход
102    if ( treePtr != NULL ) {
```

```
103       printf( "%3d", treePtr->data );
104       preOrder( treePtr->leftPtr );
105       preOrder( treePtr->rightPtr );
106    } // конец if
107 } // конец функции preOrder
108
109 // выполняет обход дерева в обратном порядке
110 void postOrder( TreeNodePtr treePtr )
111 {
112    // если дерево не пустое, выполнить обход
113    if ( treePtr != NULL ) {
114       postOrder( treePtr->leftPtr );
115       postOrder( treePtr->rightPtr );
116       printf( "%3d", treePtr->data );
117    } // конец if
118 } // конец функции postOrder
```

Пример 12.8 | Вывод программы из примера 12.7

```
The numbers being placed in the tree are:
  6  7  4 12  7dup  2  2dup  5  7dup 11
The preOrder traversal is:
  6  4  2  5  7 12 11
The inOrder traversal is:
  2  4  5  6  7 11 12
The postOrder traversal is:
  2  5  4 11 12  7  6
```

12.7.1 Функция insertNode

Функции в программе из примера рис. 12.11, используемые для создания дерева двоичного поиска и обхода его узлов, являются *рекурсивными*. Функция `insertNode` (строки 55–85) принимает *адрес дерева* и *целое число*, которое требуется сохранить в дереве. *Узлы могут вставляться в деревья двоичного поиска только как листья*. Вставка узла выполняется в три этапа:

1. Если значение `*treePtr` равно `NULL` (строка 58), создается новый узел (строка 59). Результат вызова `malloc` (адрес выделенного блока памяти) присваивается `*treePtr`, целое число, которое требуется сохранить, присваивается полю `(*treePtr)->data` (строка 63), а полям связи `(*treePtr)->leftPtr` и `(*treePtr)->rightPtr` присваивается значение `NULL` (строки 64–65), после чего управление возвращается вызывающей программе (либо функции `main`, либо предыдущему вызову `insertNode`).
2. Если значение `*treePtr` не равно `NULL` и вставляемое значение меньше `(*treePtr)->data`, производится вызов функции `insertNode` с адресом `(*treePtr)->leftPtr` (строка 74), чтобы выполнить вставку в левое поддерево узла, на который ссылается `treePtr`. Если вставляемое значение больше `(*treePtr)->data`, производится вызов функции `insertNode` с адресом `(*treePtr)->rightPtr` (строка 79), чтобы

выполнить вставку в правое поддерево узла, на который ссылается `treePtr`. *Рекурсивные вызовы* продолжаются, пока не будет встречен указатель со значением `NULL`, после чего выполняется этап 1, выполняющий *вставку нового узла*.

12.7.2 Обход дерева: функции inOrder, preOrder и postOrder

Функции `inOrder` (строки 88–96), `preOrder` (строки 99–107) и `postOrder` (строки 110–118) принимают *дерево* (то есть *указатель на корневой узел дерева*) и выполняют *обход* узлов дерева.

Функция `inOrder` выполняет обход в три этапа:

1) вызывает `inOrder` для обхода левого поддерева;
2) обрабатывает значение узла;
3) вызывает `inOrder` для обхода правого поддерева.

Значение текущего узла не обрабатывается, пока не будут обработаны значения в *левом поддереве*. В результате обхода дерева на рис. 12.13 функция `inOrder` выведет следующее:

```
6 13 17 27 33 42 48
```

В результате обхода дерева двоичного поиска функция `inOrder` выведет значения узлов в порядке *возрастания*. Процесс создания дерева двоичного поиска фактически производит сортировку данных, именно поэтому данный процесс называется **сортировкой двоичного дерева**.

Функция `preOrder` выполняет обход в три этапа:

1) обрабатывает значение узла;
2) вызывает `preOrder` для обхода левого поддерева;
3) вызывает `preOrder` для обхода правого поддерева;

Она обрабатывает узлы немедленно, в порядке их посещения. После обработки текущего узла `preOrder` вызывает себя же для обработки левого поддерева, а затем – для обработки правого поддерева. В результате обхода дерева на рис. 12.13 функция `preOrder` выведет следующее:

```
27 13 6 17 42 33 48
```

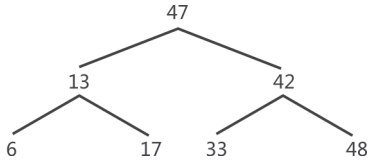


Рис. 12.13 | Дерево двоичного поиска семью узлами

Функция `postOrder` выполняет обход в три этапа:

1) вызывает `postOrder` для обхода левого поддерева;
2) вызывает `postOrder` для обхода правого поддерева;
3) обрабатывает значение узла.

Значение текущего узла не обрабатывается, пока не будут обработаны значения в дочерних поддеревьях. В результате обхода дерева на рис. 12.13 функция `postOrder` выведет следующее:

```
6 17 13 33 48 42 27
```

12.7.3 Удаление дубликатов

Дерево двоичного поиска упрощает **удаление повторяющихся элементов**. Попытка вставить повторяющийся элемент в процессе создания дерева будет немедленно распознана, потому что при этом будут выполнены те же самые переходы «влево» или «вправо», что и при вставке оригинального значения. То есть в конечном итоге при вставке повторяющегося значения будет выполнена попытка сравнить с узлом, содержащим точно такое же значение. В этой точке повторяющееся значение можно просто отбросить.

12.7.4 Поиск в двоичных деревьях

Поиск значений в двоичном дереве тоже выполняется очень быстро. Плотно упакованное дерево содержит на каждом уровне *в два раза* больше элементов, чем на предыдущем. Поэтому дерево двоичного поиска с n элементами содержит не более $\log_2 n$ уровней, откуда количество сравнений для нахождения нужного элемента или получения вывода об отсутствии искомого элемента не превышает $\log_2 n$. Это означает, например, что при поиске в (плотно упакованном) дереве двоичного поиска с 1000 элементов требуется выполнить не более 10 сравнений, потому что $2^{10} > 1000$. При поиске в (плотно упакованном) дереве двоичного поиска с 1 000 000 элементов требуется выполнить не более 20 сравнений, потому что $2^{20} > 1\,000\,000$.

12.8 Безопасное программирование на C

Глава 8 стандарта «CERT Secure C Coding Standard»

Глава 8 стандарта «CERT Secure C Coding Standard» содержит рекомендации по работе с памятью – многие из них относятся к использованию указателей и функций управления динамической памятью. За дополнительной информацией обращайтесь по адресу: `www.securecoding.cert.org`.

- MEM01-C/MEM30-C: указатели не должны оставаться неинициализированными. Именам обязательно должны присваиваться значение `NULL` или действительные адреса элементов. Вызов функции `free`, освобождающей прежде выделенную динамическую память, *не* изменя-

ет значения переданного ей указателя, поэтому указатель продолжает ссылаться на *ранее использовавшуюся* память. Повторное использование таких указателей может приводить к аварийному завершению программ и порождать уязвимости. После освобождения ранее выделенного блока динамической памяти следует немедленно присвоить указателю значение `NULL` или другой действительный адрес. Мы не следовали этой рекомендации при работе с локальными переменными-указателями, которые выходили из области видимости сразу после вызова `free`.

- MEM31-C: поведение функции `free` при попытке повторно освободить уже освобожденную память неопределенно – эта ситуация называется «уязвимостью повторного освобождения памяти». Чтобы избежать вероятности повторного освобождения одного и того же блока памяти, сразу после вызова `free` следует присвоить указателю значение `NULL` – попытка освободить память по пустому (`NULL`) указателю не дает никакого эффекта.
- MEM32-C: функция `malloc` возвращает `NULL`, когда оказывается неспособна выделить запрошенный объем памяти. Всегда проверяйте значение, возвращаемое функцией `malloc`, сравнивая его со значением `NULL`, прежде чем пытаться использовать указатель, в котором сохраняется значение, возвращаемое функцией `malloc`.

13

Препроцессор

В этой главе вы научитесь:

- пользоваться директивой `#include` при разработке больших программ;
- создавать макросы с аргументами с помощью директивы `#define`;
- применять условные директивы для определения фрагментов кода, подлежащих компиляции не во всех случаях (например, вспомогательного кода, используемого для отладки);
- выводить сообщения об ошибках в ходе компиляции;
- использовать утверждения для проверки корректности значений выражений.

Краткое содержание

13.1 Введение

Перед компиляцией программы она обрабатывается **препроцессором**. Препроцессор выполняет такие операции, как включение заголовочных файлов в компилируемый файл, определение **символических констант** и **макросов**, **условная компиляция** программного кода и **условное выполнение директив препроцессора**. Директивы препроцессора начинаются с символа **#**, при этом перед директивой препроцессора в строке могут присутствовать только пробельные символы и комментарии.

Из современных языков программирования язык C славится самым большим объемом «устаревшего кода». Он активно используется уже более четырех десятилетий. Как профессиональный программист на C вы наверняка встретитесь с кодом, написанным много лет тому назад, и столкнетесь с давно устаревшими методиками программирования. Чтобы помочь вам подготовиться к такой встрече, мы рассмотрим в этой главе некоторые из подобных приемов и познакомим вас с более современными подходами, пришедшими им на смену.

13.2 Директива препроцессора #include

Директива препроцессора `#include` широко используется в этой книге. Эта директива копирует содержимое указанного файла в место, где она находится. Директива `#include` имеет две формы использования:

```
#include <filename>
#include "filename"
```

Они отличаются лишь местом, где препроцессор будет искать указанный файл. Если имя файла `filename` заключено в угловые скобки (`<` и `>`), – используемые для **заголовочных файлов из стандартной библиотеки**, – поиск будет выполняться в каталогах, перечень которых *зависит от реализации*, обычно определяемой компилятором. Если имя файла `filename` заключено в *кавычки*, поиск файла начинается в том же каталоге, где находится компилируемый файл. Обычно эта форма директивы используется

для подключения заголовочных файлов, создаваемых программистом. Если препроцессор не сможет отыскать файл в текущем каталоге, он продолжит поиск в системных каталогах и в каталогах, определяемых компилятором.

Директива `#include` используется для подключения заголовочных файлов из стандартной библиотеки, таких как `stdio.h` и `stdlib.h` (см. табл. 5.3), и в программах, состоящих из *множества исходных файлов*, компилируемых вместе. Типичный заголовочный файл содержит объявления, общие для разных исходных файлов. Примерами таких объявлений могут служить определения *структур*, *объединений* и *перечислений*, а также *прототипы функций*.

13.3 Директива препроцессора #define: символические константы

Директива `#define` создает *символические константы* (константы, представленные последовательностями символов) и **макросы** – операции, также представленные последовательностями символов. Директива `#define` имеет следующий синтаксис:

```
#define идентификатор текст замены
```

Когда такая строка появляется в файле, все последующие вхождения *идентификатора* (находящиеся за пределами строковых литералов) перед компиляцией автоматически замещаются *текстом замены*. Например, если добавить в файл директиву:

```
#define PI 3.14159
```

препроцессор заменит все вхождения символической константы `PI` числовой константой `3.14159`. Поддержка *символических констант* дает возможность определять именованные константы и использовать их в программе. Если позднее потребуется изменить значение константы в программе, достаточно будет изменить только единственное определение в директиве `#define`. При повторной компиляции программы все вхождения константы в программе будут замещены новым значением. [*Обратите внимание*: для замены имени символической константы будет использоваться все, что находится правее этого имени.] Например, если в программе встретится объявление `#define PI = 3.14159`, препроцессор заменит все вхождения идентификатора `PI` последовательностью символов `= 3.14159`. Это может вызвать множество логических и синтаксических ошибок. По данной причине многие предпочитают использовать объявления переменных со спецификатором `const`, например

```
const double PI = 3.14159;
```

вместо директивы #define. Попытка переопределить символическую константу далее в тексте программы также вызывает ошибку во время компиляции.

Использование осмысленных имен для символических констант повышает самодокументируемость программ.

В соответствии с соглашениями в именах символических констант используются только буквы верхнего регистра и символы подчеркивания.

13.4 Директива препроцессора #define: макросы

Макрос – это идентификатор, определяемый директивой препроцессора #define. Как и символические константы, **идентификаторы макросов** замещаются **текстом замены** перед компиляцией. Макросы могут определяться с **аргументами** или без аргументов. **Макросы без аргументов** обрабатываются подобно символическим константам. В **макросах с аргументами** *формальные параметры в тексте замены замещаются фактическими значениями аргументов* при **развертывании** макроса – то есть идентификатор замещается текстом замены и фактическими аргументами, указанными в программе. Символические константы являются разновидностью макросов.

Рассмотрим следующее определение макроса с одним аргументом, вычисляющего площадь круга:

```
#define CIRCLE_AREA( x ) ( ( PI ) * ( x ) * ( x ) )
```

Когда препроцессор встретит в программе выражение CIRCLE_AREA(y), он сначала заместит в тексте замены формальный аргумент x фактическим значением y, символическую константу PI (определена выше) – ее значением, а затем развернет макрос. Например, инструкция

```
area = CIRCLE_AREA( 4 );
```

будет развернута в инструкцию:

```
area = ( ( 3.14159 ) * ( 4 ) * ( 4 ) );
```

после чего на этапе компиляции компилятор вычислит значение выражения и присвоит его переменной area. Присутствие *круглых скобок* вокруг каждого формального параметра x в тексте замены *обеспечивает правильный порядок вычисления, когда в качестве фактического аргумента передается выражение*. Например, инструкция

```
area = CIRCLE_AREA( c + 2 );
```

будет развернута в инструкцию

```
area = ( ( 3.14159 ) * ( c + 2 ) * ( c + 2 ) );
```

и вычислена *правильно*, потому что круглые скобки гарантируют правильный порядок вычислений. Если круглые скобки в определении макроса опустить, инструкция примет вид:

```
area = 3.14159 * c + 2 * c + 2;
```

и результат будет вычислен *неправильно*, так как она аналогична инструкции:

```
area = ( 3.14159 * c ) + ( 2 * c ) + 2;
```

согласно правилам предшествования операторов.

Если в тексте замены забыть заключить формальный аргумент макроса в круглые скобки, это может привести к логической ошибке.

Макрос `CIRCLE_AREA` можно заменить более безопасной функцией. Например, функция `circleArea`

```
double circleArea( double x )
{
   return 3.14159 * x * x;
}
```

производит те же вычисления, что и макрос `CIRCLE_AREA`, но аргумент функции вычисляется только один раз, при вызове функции.

В прошлом макросы часто использовались как замена функциям, чтобы избавиться от накладных расходов на вызовы функций. Современные оптимизирующие компиляторы нередко способны обеспечить встраивание функций непосредственно в код, в места их вызовов, поэтому многие программисты больше не используют макросы для этой цели. При желании можно также использовать ключевое слово `inline` *(см. приложение E).*

Ниже приводится определение макроса с двумя аргументами, вычисляющего площадь прямоугольника:

```
#define RECTANGLE_AREA( x, y ) ( ( x ) * ( y ) )
```

Когда препроцессор встретит в программе выражение `RECTANGLE_AREA (x, y)`, он заместит формальные аргументы в тексте замены фактическими значениями `x` и `y`, а затем развернет макрос, заменив идентификатор. Например, инструкция

```
rectArea = RECTANGLE_AREA( a + 4, b + 7 );
```

будет развернута в инструкцию:

```
rectArea = ( ( a + 4 ) * ( b + 7 ) );
```

Значение получившегося выражения будет вычислено во время выполнения и присвоено переменной `rectArea`.

Обычно в качестве текста замены в определениях символических констант и макросов используется весь текст, следующий за идентификатором и простирающийся до конца строки с директивой `#define`. Если текст не умещается в одной строке, его можно переносить, вставляя символ **обратного слэша** (\) в местах переноса, который будет служить признаком продолжения текста замены на следующей строке.

Объявления символических констант и макросов можно отменять с помощью директивы `#undef` препроцессора. Директива `#undef` «отменяет» определение символической константы или макроса. Таким образом, область действия символической константы или макроса простирается от определения и до директивы `#undef` или до конца файла. После отмены определения имя можно снова использовать в директиве `#define`.

Некоторые функции в стандартной библиотеке определяются как макросы, вызывающие другие библиотечные функции. Например, во многих реализациях часто можно встретить в заголовочном файле `<stdio.h>` следующий макрос:

```
#define getchar() getc( stdin )
```

В определении макроса `getchar` используется функция `getc` для получения одного символа из потока стандартного ввода. Функция `putchar`, объявленная в заголовочном файле `<stdio.h>`, и многие функции для работы с символами, объявленные в заголовочном файле `<ctype.h>`, также часто реализуются в виде макросов.

Не следует передавать макросам выражения с *побочными эффектами* (например, изменяющие значения переменных), потому что аргументы макроса могут вычисляться неоднократно. Пример такого макроса будет представлен в разделе «Безопасное программирование на C» в конце этой главы.

13.5 Условная компиляция

Поддержка **условной компиляции** позволяет управлять выполнением директив препроцессора и компиляцией программного кода. Каждая условная директива препроцессора вычисляет целочисленное значение некоторого константного выражения. Выражения приведения типов, оператор `sizeof` и константы перечислений *не* могут использоваться в директивах препроцессора.

Условная конструкция препроцессора близко напоминает инструкцию `if`. Взгляните на следующий код, обрабатываемый препроцессором:

```
#if !defined(MY_CONSTANT)
   #define MY_CONSTANT 0
#endif
```

который выясняет, *определена* ли константа `MY_CONSTANT`, то есть встречалась ли прежде директива `#define`, определяющая имя `MY_CONSTANT`. Выражение `defined(MY_CONSTANT)` возвращает `1`, если имя `MY_CONSTANT` определено, и `0` – в противном случае. Если результатом будет значение `0`, выражение `!defined(MY_CONSTANT)` вернет `1` и константа `MY_CONSTANT` будет определена директивой `#define`, следующей ниже, в противном случае она будет пропущена. Каждая конструкция `#if` должна завершаться парной ей директивой `#endif`. Существуют также директивы `#ifdef` и `#ifndef`, являющиеся сокращенными вариантами директив `#if defined(`*name*`)` и `#if !defined(`*name*`)`. Для проверки множества условий можно использовать директивы препроцессора `#elif` (эквивалент конструкции `else if` в инструкции `if`) и `#else` (эквивалент `else` в инструкции `if`). Эти директивы нередко используются для *предотвращения повторного включения заголовочных файлов*. Мы часто будем пользоваться данным приемом в части книги, посвященной языку C++.

В процессе разработки бывает удобно иметь возможность «закомментировать» фрагмент кода, чтобы предотвратить его компиляцию. Если код содержит многострочные комментарии, его нельзя будет закомментировать с помощью `/*` и `*/`, потому что многострочные комментарии не могут вкладываться друг в друга. Вместо этого можно использовать следующую условную конструкцию препроцессора:

```
#if 0
    код, который не должен компилироваться
#endif
```

Чтобы вновь включить компиляцию такого фрагмента кода, достаточно заменить `0` на `1`.

Условная компиляция часто используется в помощь для *отладки*. Многие реализации языка C включают **отладчики**, обладающие намного более мощными особенностями, чем условная компиляция. Если отладчик недоступен, часто для вывода значений переменных и слежения за потоком выполнения используется инструкция `printf`. Такие инструкции `printf` можно заключить в условные директивы препроцессора, чтобы обеспечить их компиляцию только в процессе отладки. Например, следующая конструкция:

```
#ifdef DEBUG
   printf( "Variable x = %d\n", x );
#endif
```

обеспечит компиляцию инструкции `printf`, только если выше директивы `#ifdef DEBUG` была определена символическая константа `DEBUG` (`#define DEBUG`). По окончании отладки директиву `#define` можно просто удалить из исходного файла (или закомментировать ее), и инструкции `printf`, добав-

ленные с целью отладки, будут игнорироваться компилятором. В больших программах может оказаться желательным определить несколько разных символических констант, управляющих условной компиляцией разных фрагментов в исходных файлах. Многие компиляторы дают возможность определять и отменять определение символических констант с помощью флагов компилятора, чтобы избавить программиста от необходимости изменять исходный код.

Часто программисты допускают ошибку, вставляя условно компилируемые инструкции `printf` *для нужд отладки туда, где ожидается единственная инструкция. В таких случаях условно компилируемая инструкция должна заключаться в составную инструкцию, чтобы после компиляции программы с отладочными инструкциями направление потока выполнения программы не изменялось.*

13.6 Директивы препроцессора #error и #pragma

Директива `#error`

```
#error лексемы
```

выводит сообщение, текст которого зависит от реализации, включающее *лексемы*, объявленные в директиве. Лексемы – это последовательность символов, разделенных пробелами. Например, директива

```
#error 1 - Out of range error
```

содержит 6 лексем. В некоторых системах, когда встречается директива `#error`, указанные в ней лексемы выводятся в составе сообщения об ошибке, препроцессор прекращает свою работу, и программа не компилируется.

Директива `#pragma`

```
#pragma лексемы
```

выполняет некоторое действие, *зависящее от реализации*. Программы, которые не распознаются реализацией, просто игнорируются. За более полной информацией о директивах `#error` и `#pragma` обращайтесь к документации для своего компилятора C.

13.7 Операторы # и

Операторы препроцессора # и ## определяются стандартом языка C. Оператор # обеспечивает замену текста лексемы строкой, заключенной в кавычки. Рассмотрим следующее определение макроса:

```
#define HELLO(x) puts( "Hello, " #x );
```

Если вставить в текст программы обращение к макросу `HELLO(John)`, оно будет развернуто в инструкцию

```
puts( "Hello, " "John" );
```

Выражение `#x` в тексте замены макроса будет замещено строкой `"John"`. Строки, отделяемые друг от друга пробелами, объединяются на этапе обработки исходного кода препроцессором, поэтому предыдущая инструкция будет фактически преобразована в следующую:

```
puts( "Hello, John" );
```

Оператор `#` должен использоваться в макросах с аргументами, потому что операндом оператора `#` является аргумент макроса.

Оператор `##` объединяет две лексемы. Взгляните на следующее определение макроса:

```
#define TOKENCONCAT(x, y) x ## y
```

Когда препроцессор встретит идентификатор `TOKENCONCAT` в тексте программы, его аргументы будут объединены и использованы для подстановки вместо идентификатора. Например, выражение `TOKENCONCAT(O, K)` будет замещено лексемой `OK`. Оператор `##` имеет два операнда.

13.8 Номера строк

Директива препроцессора `#line` вызывает перенумерацию строк исходного кода, начиная с указанного целочисленного значения. Так, директива

```
#line 100
```

присвоит следующей за ней строке исходного кода порядковый номер `100`. В директиву `#line` можно также включить имя исходного файла, например директива

```
#line 100 "file1.c"
```

указывает, что нумерация строк, следующих ниже, начнется с числа 100, и во всех последующих сообщениях компилятора будет фигурировать имя файла `"file1.c"`. Обычно эта директива используется с целью сделать предупреждения компилятора и сообщения о синтаксических ошибках более осмысленными. Данная директива не вызывает вставку номеров строк в исходный файл.

13.9 Предопределенные символические константы

Стандарт языка С определяет несколько предопределенных символических констант, часть из которых перечислена в табл. 13.1 – полный перечень можно найти в разделе 6.10.8 стандарта языка С. Идентификаторы всех предопределенных символических констант начинаются и оканчиваются *двумя* символами подчеркивания. Эти идентификаторы, а также идентификатор `defined` (см. раздел 13.5) не могут использоваться в директивах `#define` и `#undef`.

Таблица 13.1 | Некоторые предопределенные символические константы

Символическая константа	Описание
`__LINE__`	Номер текущей строки в исходном коде (целочисленная константа)
`__FILE__`	Предполагаемое имя исходного файла (строка)
`__DATE__`	Дата компиляции исходного файла (строка в формате `"Mmm dd yyyy"`, такая как `"Jan 19 2002"`)
`__TIME__`	Время компиляции исходного файла (строка в формате `"hh:mm:ss"`)
`__STDC__`	Имеет значение `1`, если компилятор соответствует стандарту языка С

13.10 Утверждения

Макрос `assert` – объявленный в заголовочном файле `<assert.h>` – проверяет значение выражения во время выполнения. Если выражение возвращает ложное (`0`) значение, `assert` выводит сообщение об ошибке и вызывает функцию `abort` (библиотека универсальных утилит – `<stdlib.h>`) для завершения программы. Это очень удобный *отладочный инструмент*, позволяющий проверять корректность значений переменных. Например, допустим, что переменная `x` никогда не должна принимать значение больше `10`. Для проверки значения переменной `x` можно задействовать утверждение и с его помощью выводить сообщение об ошибке, если значение `x` превысит установленный порог:

```
assert( x <= 10 );
```

Если `x` окажется больше `10`, инструкция выше выведет сообщение об ошибке с номером строки и именем файла и *завершит* программу. Благодаря этому вы сможете сконцентрировать свое внимание на строках, предшествующих этой инструкции, чтобы отыскать ошибку. Если определена символическая константа `NDEBUG`, утверждения будут игнорироваться ком-

пилятором. То есть когда надобность в утверждениях отпадет, достаточно просто вставить директиву

```
#define NDEBUG
```

в файл программы, чтобы отменить все проверки с помощью макроса `assert`.

Утверждения не могут служить заменой обработки ошибок, возникающих при нормальном выполнении программы. И их следует использовать только для поиска логических ошибок в процессе разработки.

[*Обратите внимание*: новый стандарт языка C включает такую особенность, как `_Static_assert`, которая фактически является разновидностью утверждений, проверяемых на этапе компиляции, которые генерируют сообщения об ошибках во время компиляции, если проверяемое утверждение оказывается ложно. Мы будем рассматривать `_Static_assert` в приложении E.]

13.11 Безопасное программирование на C

Макрос `CIRCLE_AREA`, объявленный в разделе 13.4 как

```
#define CIRCLE_AREA( x ) ( ( PI ) * ( x ) * ( x ) )
```

считается *небезопасным макросом*, потому вычисляет свой аргумент `x` более одного раза. Это может вызывать весьма трудноуловимые ошибки. Если при вычислении аргумента макроса производятся какие-либо *побочные эффекты*, такие как инкремент переменной или вызов функции, изменяющей значение переменной, эти побочные эффекты будут воспроизводиться *несколько* раз.

Например, если вызвать макрос `CIRCLE_AREA`, как показано ниже:

```
result = CIRCLE_AREA( ++radius );
```

в результате будет скомпилирована следующая инструкция:

```
result = ( ( 3.14159 ) * ( ++radius ) * ( ++radius ) );
```

которая увеличивает переменную `radius` *дважды*. Кроме того, результат предыдущей инструкции *не определен*, потому что C позволяет модифицировать переменную в инструкции *только один раз*. В вызове функции аргумент вычисляется *лишь один раз перед* передачей функции. Поэтому вместо небезопасных макросов всегда лучше использовать функции.

14

Разное

В этой главе вы узнаете:

- как перенаправить ввод, чтобы данные поступали не с клавиатуры, а из файла;
- как перенаправить вывод, чтобы данные не выводились на экран, а сохранялись в файле;
- как писать функции, принимающие переменное количество аргументов;
- как обрабатывать аргументы командной строки;
- как присваивать определенные типы числовым константам;
- как обрабатывать в программе внешние асинхронные события;
- как динамически распределять массивы и изменять объем блока динамической памяти, выделенного прежде.

14.1 Введение

В этой главе рассматривается несколько дополнительных тем. Многие особенности, обсуждаемые здесь, являются характерными для отдельных операционных систем.

14.2 Перенаправление ввода/вывода

В приложениях командной строки стандартный ввод обычно связан с *клавиатурой*, а стандартный вывод – с *экраном*. В большинстве операционных систем – Linux/UNIX, Mac OS X и Windows, в частности, – поддерживается возможность **перенаправления** стандартного ввода и стандартного вывода так, что информация будет извлекаться *из файла* или сохраняться *в файле* соответственно. Обе формы перенаправления поддерживаются средствами для работы с файлами, имеющимися в стандартной библиотеке.

Существует несколько способов перенаправления ввода/вывода из командной строки – то есть в окне **Command Prompt** (**Командная строка**) в Windows, в командной оболочке в Linux или в окне **Terminal** (**Терминал**) в Mac OS X. Представьте, что у вас имеется выполняемый файл `sum` (в Linux/UNIX) с программой, которая вводит целые числа по одному, пока не будет введен признак конца файла, и затем выводит результат их сложения. Обычно пользователь вводит целые числа с клавиатуры и в конце нажимает комбинацию клавиш, чтобы ввести признак конца файла. Воспользовавшись поддержкой перенаправления, исходные числа можно сохранить в файле и затем передать этот файл программе для вычисления суммы. Например, если предположить, что исходные данные хранятся в файле `input`, тогда следующая команда:

```
$ sum < input
```

запустит программу `sum`; **символ перенаправления ввода** (`<`) указывает, что данные из файла `input` будут поступать на стандартный ввод программы. Перенаправление в Windows выполняется идентично.

Символ `$` считается типичным приглашением к вводу в командных оболочках Linux/UNIX (в некоторых системах используется символ `%` или другой). Перенаправление, как было показано выше, является особенностью операционной системы, а не языка C.

Второй способ перенаправления ввода – организация **конвейера**. Конвейер (`|`) обеспечивает передачу вывода одной программы на ввод другой. Допустим, что у нас имеется программа `random`, которая выводит последовательность случайных целых чисел; вывод программы `random` можно передать «по конвейеру» непосредственно программе `sum`:

```
$ random | sum
```

Эта команда обеспечит вычисление суммы случайных целых чисел, поставляемых программой `random`. Объединение программ в конвейеры выполняется одинаково и в Linux/UNIX, и в Windows.

Поток стандартного вывода также можно перенаправить в файл с помощью **символа перенаправления вывода** (`>`). Например, чтобы сохранить вывод программы `random` в файле `out`, можно выполнить такую команду:

```
$ random > out
```

Наконец, существует возможность добавить вывод программы в конец существующего файла, воспользовавшись **символом добавления вывода в конец** (`>>`). Например, чтобы добавить вывод программы `random` в конец файла `out`, созданного предыдущей командой, можно выполнить такую команду:

```
$ random >> out
```

14.3 Функции с переменным количеством аргументов

В языке C поддерживается возможность создавать функции, способные принимать переменное количество аргументов. Большинство программ в этой книге используют функцию `printf` из стандартной библиотеки, которая, как вы уже знаете, может принимать любое количество аргументов. Как минимум функция `printf` требует передачи строки в первом аргументе, но может принимать любое количество дополнительных аргументов. Функция `printf` имеет следующий прототип:

```
int printf( const char *format, ... );
```

Многоточие (`...`) в прототипе функции указывает, что она принимает *переменное количество аргументов любого типа*. Многоточие всегда должно следовать последним в списке параметров.

Макросы и определения для использования в **заголовках функций с переменным количеством аргументов** находятся в заголовочном файле `<stdarg.h>` (табл. 14.1) и покрывают все потребности, возникающие при создании функций со **списками аргументов переменной длины**. В примере 14.1 демонстрируется реализация функции `average` (строки 25–40), которая принимает переменное количество аргументов. В первом аргументе `average` всегда принимает количество значений, по которым требуется вычислить среднее.

Таблица 14.1 | Макросы и определения для использования в заголовках функций с переменным количеством аргументов (stdarg.h)

Идентификатор	Описание
`va_list`	Тип, пригодный для хранения информации, необходимой макросам `va_start`, `va_arg` и `va_end`. Чтобы иметь возможность обращаться к списку аргументов переменной длины, необходимо определить объект типа `va_list`
`va_start`	Макрос, вызываемый перед первым обращением к списку аргументов переменной длины. Этот макрос инициализирует объект типа `va_list`, используемый макросами `va_arg` и `va_end`
`va_arg`	Макрос, разворачивающийся в значение следующего аргумента в списке, – тип значения определяется вторым аргументом макроса. Каждый вызов макроса `va_arg` изменяет объект типа `va_list` так, что после вызова он ссылается на следующий аргумент в списке
`va_end`	Макрос, упрощающий нормальный возврат из функции со списком аргументов переменной длины, ссылку на который возвращает макрос `va_start`

Пример 14.1 | Использование списка аргументов переменной длины

```
 1 // Пример 14.1: fig14_02.c
 2 // Использование списка аргументов переменной длины
 3 #include <stdio.h>
 4 #include <stdarg.h>
 5
 6 double average( int i, ... ); // прототип
 7
 8 int main( void )
 9 {
10    double w = 37.5;
11    double x = 22.5;
12    double y = 1.7;
13    double z = 10.2;
14
15    printf( "%s%.1f\n%s%.1f\n%s%.1f\n%s%.1f\n\n",
16       "w = ", w, "x = ", x, "y = ", y, "z = ", z );
17    printf( "%s%.3f\n%s%.3f\n%s%.3f\n",
18       "The average of w and x is ", ,
```

```
19       "The average of w, x, and y is ", average( 2, w, x ),
20       "The average of w, x, y, and z is ",
21       average( 4, w, x, y, z ) );
22 } // конец main
23
24 // вычисляет среднее
25 double average( int i, ... )
26 {
27    double total = 0; // инициализировать сумму total
28    int j;            // счетчик выбранных аргументов
29    va_list ap;       // хранит информацию для va_start и va_end
30
31    va_start( ap, i );// инициализировать объект va_list
32
33    // обработать список аргументов переменной длины
34    for ( j = 1; j <= i; ++j ) {
35       total += va_arg( ap, double );
36    } // конец for
37
38    va_end( ap );     // очистить список аргументов переменной длины
39    return total / i; // вернуть среднее
40 } // конец функции average
```

```
w = 37.5
x = 22.5
y = 1.7
z = 10.2

The average of w and x is 30.000
The average of w, x, and y is 20.567
The average of w, x, y, and z is 17.975
```

Функция `average` (строки 25–40) использует все определения и макросы из заголовочного файла `<stdarg.h>`. Объект `ap` типа `va_list` (строка 29) используется макросами `va_start`, `va_arg` и `va_end` для обработки списка аргументов переменной длины. Функция начинается с вызова макроса `va_start` (строка 31), который выполняет инициализацию объекта `ap` для последующего его использования в макросах `va_arg` и `va_end`. Этот макрос принимает два аргумента – объект `ap` и идентификатор последнего аргумента, непосредственно *предшествующего* многоточию, – в данном случае `i` (`va_start` использует идентификатор `i` для определения начала списка аргументов переменной длины). Затем функция `average` в цикле складывает значения аргументов из списка, сохраняя сумму в переменной `total` (строки 34–36). Значение, прибавляемое к сумме `total`, извлекается из списка с помощью макроса `va_arg`. Макрос `va_arg` принимает два аргумента – объект `ap` и ожидаемый *тип* значения в списке аргументов – в данном случае `double`, и возвращает значение аргумента. Чтобы обеспечить нормальный возврат в вызывающую программу, функция `average` вызывает макрос `va_end` (строка 38) с объектом `ap`. В заключение `average` вычисляет среднее значение и возвращает его функции `main`.

Если поместить многоточие в середину списка параметров функции, это вызовет синтаксическую ошибку – многоточие должно замыкать список параметров.

Кто-то из читателей может задаться вопросом, как функции `printf` и `scanf` узнают, какой тип передать макросу `va_arg`. Ответ на этот вопрос прост: они сканируют строку формата, вычленяют из нее спецификаторы преобразований и по ним определяют типы дополнительных аргументов.

14.4 Использование аргументов командной строки

Во многих системах поддерживается возможность передачи аргументов командной строки в функцию `main` объявлением параметров `int argc` и `char *argv[]` в заголовке функции `main`. В параметре `argc` функции `main` передается количество аргументов командной строки, а в параметре `argv` – массив строк с фактическими значениями аргументов. Обычно аргументы командной строки используются для передачи программе ключей, определяющих некоторые особенности ее поведения, и имен файлов.

Программа в примере 14.2 копирует содержимое одного файла в другой по одному символу. Если предположить, что выполняемый файл программы называется `mycopy`, тогда вызов этой программы в системе Linux/UNIX будет выглядеть так:

```
$ mycopy input output
```

Из этой команды видно, что содержимое файла `input` копируется в файл `output`. Когда программа запускается, она проверяет значение `argc`. Если оно оказывается не равно `3` (в первом аргументе программам всегда передается имя выполняемого файла программы), программа выводит сообщение об ошибке и завершается. В противном случае она получает массив `argv` со строками `"mycopy"`, `"input"` и `"output"`. Второй и третий элементы массива (соответствующие первому и второму аргументам командной строки) интерпретируются программой как имена файлов. Файлы открываются с помощью функции `fopen`. Если оба файла удалось благополучно открыть, программа начинает читать символы из одного файла `input` и записывать их в файл `output`, пока не встретит признак конца файла `input`, после чего программа завершается. В результате создается точная копия файла `input` (если в процессе выполнения не возникнет ошибок). За дополнительной информацией об аргументах командной строки обращайтесь к документации с описанием вашей системы. [*Обратите внимание*: в среде разработки Visual C++ аргументы командной строки можно определить, если щелкнуть правой кнопкой мыши на имени проекта в **Solution Explorer** (**Обозреватель**

решения), выбрать пункт контекстного меню **Properties** (**Свойства**), распахнуть ветку **Configuration Properties** (**Параметры конфигурации**), выбрать пункт **Debugging** (**Отладка**) и ввести аргументы в текстовое поле **Command Arguments** (**Аргументы командной строки**).]

Пример 14.2 | Использование аргументов командной строки

```
// Пример 14.2: fig14_03.c
// Использование аргументов командной строки
#include <stdio.h>

int main( int argc, char *argv[] )
{
   FILE *inFilePtr;  // указатель на файл input
   FILE *outFilePtr; // указатель на файл output
   int c;            // хранит символ, прочитанный из исходного файла

   // проверить количество аргументов командной строки
   if( argc != 3 ){
      puts( "Usage: mycopy infile outfile" );
   } // конец if
   else {
      // если файл input удалось открыть
      if ( ( inFilePtr = fopen(argv[ 1 ], "r" ) ) != NULL ) {
         // если файл output удалось открыть
         if ( ( outFilePtr = fopen(argv[ 2 ], "w" ) ) != NULL ) {
            // читать и выводить символы по одному
            while ( ( c = fgetc( inFilePtr ) ) != EOF ) {
               fputc( c, outFilePtr );
            } // конец while

            fclose( outFilePtr ); // закрыть файл output
         } // конец if
         else { // файл output не удалось открыть
            printf( "File \"%s\" could not be opened\n", argv[ 2 ] );
         } // конец else

         fclose( inFilePtr ); // закрыть файл input
      } // конец if
      else { // файл input не удалось открыть
         printf( "File \"%s\" could not be opened\n", argv[ 1 ] );
      } // конец else
   } // конец else
} // конец main
```

14.5 Замечания о компиляции программ из нескольких исходных файлов

Существует возможность компилировать программы, состоящие из нескольких файлов с исходными текстами. Однако есть несколько важных аспектов, которые необходимо учитывать при создании таких программ. Например, определение функции должно целиком находиться в одном файле – его нельзя разбить на два файла или более.

В главе 5 мы познакомились с понятиями класса хранения и области видимости и узнали, что переменные, объявленные *за пределами* функций, являются *глобальными переменными*. Глобальные переменные доступны из любых функций, объявленных в том же файле, ниже объявления переменной. Глобальные переменные также доступны функциям, объявленным в других файлах, но при этом их необходимо явно объявлять в каждом файле, где они используются. Например, чтобы получить возможность обращаться к глобальной переменной `flag` в другом файле, ее необходимо объявить в этом файле, как показано ниже:

```
extern int flag;
```

В этом объявлении используется спецификатор класса хранения `extern`, указывающий, что переменная `flag` объявляется *либо ниже в этом же файле, либо в другом файле*. Компилятор сообщит компоновщику о неразрешенной ссылке на внешнюю переменную `flag`. Если компоновщик найдет подходящее определение глобальной переменной, он подставит ее адрес в машинную инструкцию, обращающуюся к этой переменной. Если же компоновщик не найдет определение переменной `flag`, он выведет сообщение об ошибке и прервет процедуру создания выполняемого файла. Любой идентификатор, объявляемый в области видимости файла, автоматически получает класс хранения `extern`.

Желательно избегать использования глобальных переменных, если только на первом месте не стоит обеспечение максимальной производительности приложения, потому что они нарушают принцип наименьших привилегий.

Подобно тому, как объявления `extern` позволяют распространить область действия *глобальных переменных* на несколько файлов, входящих в состав программы, *объявления прототипов функций дают возможность использовать эти функции за пределами файлов, где они определены* (указывать спецификатор `extern` в прототипах функций не требуется). Просто включите прототип функции в каждый файл, где она вызывается, и скомпилируйте файлы вместе (см. раздел 13.2). Прототип функции указывает компилятору, что соответствующая функция определена либо ниже *в этом же* файле, либо *в другом* файле. И снова компилятор *не* будет пытаться разрешить ссылки на такие функции, оставив решение этой задачи компоновщику. Если компоновщик не встретит подходящее определение функции, он выведет сообщение об ошибке и прервет процедуру создания выполняемого файла.

Как пример использования прототипов функций для расширения области их видимости представьте любую программу, содержащую директиву препроцессора `#include <stdio.h>`, которая подключает файл с прототипами таких функций, как `printf` и `scanf`. Благодаря этому другие функции в файле получают возможность использовать `printf` и `scanf` для решения

своих задач. Функции `printf` и `scanf` определены в других файлах. Нам совсем *необязательно* знать, *в каких именно файлах* они определены. Мы просто используем их в своих программах. Ссылки на эти функции разрешаются компоновщиком на этапе сборки программы. Это дает нам возможность использовать функции из стандартной библиотеки.

Распределение реализации программы по нескольким файлам способствует повторному использованию имеющегося программного кода и выработке удачных архитектурных решений. Функции могут оказаться достаточно универсальными, чтобы использоваться во множестве приложений. В таких случаях определения этих функций следует сохранять в отдельных исходных файлах, и для каждого исходного файла нужно создать соответствующий заголовочный файл с прототипами функций. Подобный подход позволяет программистам повторно использовать тот же самый программный код, включая соответствующие заголовочные файлы и компилируя свои приложения совместно с соответствующими исходными файлами.

Поддерживается также возможность ограничить область видимости глобальных переменных или функций границами файла, в котором они определяются. Когда к глобальной переменной или функции применяется спецификатор класса хранения `static`, это предотвращает возможность использования таких переменных и функций за пределами данного файла. Это называется **внутренним связыванием** (internal linkage). Глобальные переменные и функции, определяемые без спецификатора `static`, поддерживают **внешнее связывание** (external linkage) – они доступны в других файлах, при наличии в этих файлах соответствующих объявлений.

Определение глобальной переменной

```
static const double PI = 3.14159;
```

создаст неизменяемую переменную `PI` типа `double`, инициализированную значением `3.14159` и доступную *только* функциям в файле, где объявлена эта переменная.

Спецификатор `static` часто применяется при создании вспомогательных функций, используемых лишь в пределах данного файла. Если функция не нужна за пределами текущего файла, в соответствии с принципом наименьших привилегий ее следует объявить с классом `static`. Если функция определяется *до* ее использования в файле, спецификатор `static` следует применить к определению функции, в противном случае – к прототипу.

Компиляция больших программ из множества исходных файлов становится порой весьма утомительным занятием: если в каком-то одном файле были произведены небольшие изменения, приходится перекомпилировать всю программу. Во многих системах существуют специальные утилиты, которые автоматизируют эту работу и компилируют только изменившиеся файлы. В системах Linux/UNIX такая утилита называется `make`. Инструкции

по компиляции и компоновке программы передаются утилите `make` в файле с именем `Makefile`. Интегрированные среды разработки, такие как Eclipse™ и Microsoft® Visual C++®, включают похожие утилиты.

14.6 Завершение выполнения программ с помощью функций exit и atexit

Библиотека универсальных утилит (`<stdlib.h>`) предоставляет несколько способов завершения программы, кроме обычного возврата из функции `main`. Одним из таких способов является вызов функции `exit`. Она часто используется для принудительного завершения программы при обнаружении ошибки ввода или когда программа не может открыть файл, который требуется обработать. Функция `atexit` позволяет зарегистрировать функцию, которая должна быть вызвана в случае нормального завершения программы, то есть либо когда программа завершается, достигнув конца функции `main`, либо когда вызывается функция `exit`.

Функция `atexit` принимает указатель на функцию (то есть *имя функции*). *Функции, вызываемые перед завершением программы, не должны иметь параметров и возвращаемых значений.*

Функция `exit` принимает один аргумент. Обычно ей передается символическая константа `EXIT_SUCCESS` или `EXIT_FAILURE`. Если `exit` вызывается с константой `EXIT_SUCCESS`, вызывающему окружению возвращается соответствующий код благополучного завершения. Если `exit` вызывается с константой `EXIT_FAILURE`, окружению возвращается код завершения с ошибкой. Когда вызывается функция `exit`, автоматически вызываются все функции, зарегистрированные прежде с помощью функции `atexit`, в порядке, *обратном* порядку их регистрации, все потоки данных, открытые в программе, очищаются и закрываются, и управление возвращается среде выполнения.

Программа в примере 14.3 демонстрирует применение функций `exit` и `atexit`. Программа предлагает пользователю определить, как она должна завершиться – в результате вызова функции `exit` или по достижении конца функции `main`. В обоих случаях по завершении программы вызывается функция `print`.

Пример 14.3 | Демонстрация применения функций exit и atexit

```
1 // Пример 14.3: fig14_04.c
2 // Демонстрация применения функций exit и atexit
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 void print( void ); // прототип
7
8 int main( void )
```

```
 9 {
10    int answer; // пункт меню, выбранный пользователем
11
12    atexit( print ); // зарегистрировать функцию print
13    puts( "Enter 1 to terminate program with function exit"
14       "\nEnter 2 to terminate program normally" );
15    scanf( "%d", &answer );
16
17    // вызвать exit, если пользователь ввел 1
18    if ( answer == 1 ) {
19       puts( "\nTerminating program with function exit" );
20       exit( EXIT_SUCCESS );
21    } // конец if
22
23    puts( "\nTerminating program by reaching the end of main" );
24 } // конец main
25
26 // выводит сообщение перед завершением
27 void print( void )
28 {
29    puts( "Executing function print at program "
30       "termination\nProgram terminated" );
31 } // конец функции print
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
1

Terminating program with function exit
Executing function print at program termination
Program terminated
```

```
Enter 1 to terminate program with function exit
Enter 2 to terminate program normally
2

Terminating program by reaching the end of main
Executing function print at program termination
Program terminated
```

14.7 Окончания в литералах целых и вещественных чисел

В языке C имеется возможность добавлять *специальные окончания* в литералы целых и вещественных чисел, чтобы точно указывать типы значений литералов. (В стандарте языка C литеральные значения называются константами). Если литерал целого числа *не имеет* окончания, его тип подбирается, исходя из объема памяти, необходимого для хранения данного значения (сначала проверяется совместимость с типом `int`, затем с типом `long int`, потом `unsigned long int` и т. д.). Литералы вещественных чисел *без* окончаний автоматически получают тип `double`.

В литералах целых чисел поддерживаются следующие окончания: `u` и `U` – соответствуют типу `unsigned int`, `l` и `L` – типу `long int`, `ll` и `LL` – типу `long long int`. Окончания `u` и `U` можно объединять с окончаниями, соответствующими типам `long int` и `long long int` для создания литералов беззнаковых целочисленных типов большого размера. Следующие литералы имеют тип `unsigned int`, `long int`, `unsigned long int` и `unsigned long long int` соответственно:

```
174u
8358L
28373ul
9876543210llu
```

В литералах вещественных чисел поддерживаются следующие окончания: `f` и `F` – соответствуют типу `float`, а `l` и `L` – типу `long double`. Следующие литералы имеют тип `float` и `long double` соответственно:

```
1.28f
3.14159L
```

14.8 Обработка сигналов

Внешнее асинхронное **событие**, или **сигнал**, может вызвать преждевременное завершение программы. К числу событий относятся также **прерывания** (ввод комбинации **<Ctrl> c** в Linux/UNIX или Windows), появление **недопустимой инструкции** в машинном коде, **нарушение прав доступа к памяти**, принудительное завершение со стороны операционной системы и **исключение в операции с плавающей запятой** (floating-point exception), например деление на ноль или умножение на очень большое вещественное значение. Библиотека функций для работы с сигналами (`<signal.h>`) дает возможность перехватывать неожиданные события с помощью функции `signal`. Функция `signal` принимает два аргумента – целочисленный *номер сигнала* и *указатель на функцию обработки сигнала*. Сигналы могут генерироваться вызовом функции `raise`, которая принимает целочисленный номер сигнала. В табл. 14.2 приводится краткий перечень *стандартных сигналов*, объявленных в заголовочном файле `<signal.h>`.

Таблица 14.2 | Стандартные сигналы (signal.h)

Сигнал	Описание
`SIGABRT`	Аварийное завершение программы (например, в результате вызова функции `abort`)
`SIGFPE`	Ошибочная арифметическая операция, такая как деление на ноль или операция, в результате которой произошло переполнение
`SIGILL`	Обнаружена недопустимая инструкция
`SIGINT`	Сигнал интерактивного привлечения внимания
`SIGSEGV`	Попытка обратиться к памяти, которая не выделялась программой
`SIGTERM`	Программе передан запрос на завершение

Программа в примере 14.4 демонстрирует применение функции `signal` для *перехвата* сигнала `SIGINT`. В строке 15 она вызывает функцию `signal` и передает ей номер сигнала `SIGINT` и указатель на функцию `signalHandler` (не забывайте, что имя функции интерпретируется как указатель на начало функции). Когда в программу поступает сигнал `SIGINT`, управление передается функции `signalHandler`, которая выводит сообщение и дает пользователю продолжить нормальное выполнение программы. Если пользователь решит продолжить работу программы, обработчик повторно инициализируется вызовом функции `signal` и возвращает управление в точку в программе, где был обнаружен сигнал. Для имитации сигнала `SIGINT` в этой программе используется функция `raise` (строка 24). Программа выбирает случайное число в диапазоне от 1 до 50 и, если это число равно 25, вызывает функцию `raise`, чтобы сгенерировать сигнал. Обычно сигнал `SIGINT` генерируется за пределами программы. Например, ввод комбинации **<Ctrl> c** в ходе выполнения программы в Linux/UNIX или Windows генерирует сигнал `SIGINT`, который *завершает* программу. Прием обработки сигналов можно использовать для перехвата сигнала `SIGINT` и предотвращения преждевременного завершения программы.

Пример 14.4 | Прием обработки сигналов

```
 1 // Пример 14.4: fig14_06.c
 2 // Прием обработки сигналов
 3 #include <stdio.h>
 4 #include <signal.h>
 5 #include <stdlib.h>
 6 #include <time.h>
 7
 8 void signalHandler( int signalValue ); // прототип
 9
10 int main( void )
11 {
12    int i; // счетчик цикла
13    int x; // переменная для хранения случайных чисел
14
15    signal( SIGINT, signalHandler ); // зарегистрировать обработчик сигнала
16    srand( time( NULL ) );
17
18    // вывести числа от 1 до 100
19    for ( i = 1; i <= 100; ++i ) {
20       x = 1 + rand() % 50; // сгенерировать случайное число
21
22       // сгенерировать сигнал SIGINT, если x == 25
23       if ( x == 25 ) {
24          raise( SIGINT );
25       } // конец if
26
27       printf( "%4d", i );
28
29       // вывести \n после каждого 10-го числа
30       if ( i % 10 == 0 ) {
31          printf( "%s", "\n" );
```

```
32       } // конец if
33    } // конец for
34 } // конец main
35
36 // обрабатывает сигнал
37 void signalHandler( int signalValue )
38 {
39    int response; // ответ пользователя (1 или 2)
40
41    printf( "%s%d%s\n%s",
42       "\nInterrupt signal ( ", signalValue, " ) received.",
43       "Do you wish to continue ( 1 = yes or 2 = no )? " );
44
45    scanf( "%d", &response );
46
47    // проверить допустимость ответа
48    while ( response != 1 && response != 2 ) {
49       printf( "%s", "( 1 = yes or 2 = no )? " );
50       scanf( "%d", &response );
51    } // конец while
52
53    // определить, не пора ли завершить программу
54    if ( response == 1 ) {
55       // зарегистрировать обработчик, чтобы перехватить очередной SIGINT
56       signal( SIGINT, signalHandler );
57    } // конец if
58    else {
59       exit( EXIT_SUCCESS );
60    } // конец else
61 } // конец функции signalHandler
```

```
   1   2   3   4   5   6   7   8   9  10
  11  12  13  14  15  16  17  18  19  20
  21  22  23  24  25  26  27  28  29  30
  31  32  33  34  35  36  37  38  39  40
  41  42  43  44  45  46  47  48  49  50
  51  52  53  54  55  56  57  58  59  60
  61  62  63  64  65  66  67  68  69  70
  71  72  73  74  75  76  77  78  79  80
  81  82  83  84  85  86  87  88  89  90
  91  92  93
Interrupt signal ( 2 ) received.
Do you wish to continue ( 1 = yes or 2 = no )? 1
  94  95  96
Interrupt signal ( 2 ) received.
Do you wish to continue ( 1 = yes or 2 = no )? 2
```

14.9 Динамическое выделение памяти: функции calloc и realloc

В главе 12 мы познакомились с возможностью динамического выделения памяти с помощью функции `malloc`. Как отмечалось в главе 12, массивы обеспечивают более высокую скорость сортировки, поиска и доступа к данным,

чем связанные списки. Однако массивы, как правило, являются **статическими структурами данных**. Библиотека универсальных утилит (`<stdlib.h>`) предоставляет еще две функции динамического выделения памяти – `calloc` и `realloc`. Эти функции можно использовать для создания динамических массивов и изменения их размеров. Как было показано в главе 7, к указателю на массив можно применять оператор индексирования, как к обычному массиву. То есть указатель на непрерывную область памяти, выделенную с помощью функции `calloc`, можно интерпретировать как массив. Функция `calloc` динамически выделяет память для массива. Она имеет следующий прототип:

```
void *calloc( size_t nmemb, size_t size );
```

Два ее аргумента представляют *количество элементов* в массиве (`nmemb`) и *размер* одного элемента (`size`). Функция `calloc` также инициализирует элементы массива нулевыми значениями. Функция возвращает указатель на выделенный блок памяти или `NULL`, если память *не была* выделена. Главное отличие между функциями `malloc` и `calloc` – в том, что `calloc` *очищает выделенную память*, а `malloc` – *нет*.

Функция `realloc` *изменяет объем памяти*, выделенной предыдущим вызовом `malloc`, `calloc` или `realloc`. Содержимое области памяти при этом не изменяется, при условии что ее объем увеличивается. В противном случае содержимое остается неизменным только до установленной границы нового блока. Функция `realloc` имеет следующий прототип:

```
void *realloc( void *ptr, size_t size );
```

Она принимает два аргумента: указатель на оригинальный блок памяти (`ptr`) и новый размер блока (`size`). Если в аргументе `ptr` передать `NULL`, `realloc` будет действовать идентично функции `malloc`. Если аргумент `ptr` не равен `NULL` и аргумент `size` больше нуля, `realloc` попытается *выделить новый блок памяти* указанного размера. Если новый блок *не может* быть выделен, блок памяти, на который ссылается аргумент `ptr`, останется без изменений. Функция `realloc` возвращает либо указатель на вновь выделенный блок памяти, либо `NULL`, чтобы показать, что память не была выделена.

Избегайте выделять блоки памяти нулевого размера с помощью функций `malloc`, `calloc` и `realloc`.

14.10 Безусловные переходы с помощью goto

На протяжении всей книги мы подчеркивали важность приемов структурного программирования для создания надежного программного обеспечения, упрощающих отладку, сопровождение и расширение. Однако в некоторых

ситуациях производительность оказывается важнее строгого следования принципам структурного программирования. В таких случаях допускается использовать приемы неструктурного программирования. Например, мы можем использовать инструкцию `break`, чтобы прервать выполнение цикла до того, как условие продолжения примет ложное значение. Это поможет сэкономить время на нескольких итерациях цикла, если задачу удалось решить *раньше*, чем предполагалось.

Еще одним представителем неструктурного программирования является инструкция безусловного перехода `goto`. Результатом выполнения инструкции `goto` будет передача управления первой инструкции, следующей за **меткой**, указанной в инструкции `goto`. Метка – это идентификатор, за которым следует двоеточие. Метка должна находиться внутри той же функции, что и инструкция `goto`, ссылающаяся на нее.

Программа в примере 14.5 демонстрирует применение инструкций `goto` для организации цикла, выполняющего 10 итераций и выводящего значение счетчика. После инициализации счетчика значением 1 в строке 11 выполняется проверка значения `count`, чтобы определить, не превысило ли оно значение `10` (метка `start:` пропускается, потому что метки сами по себе не выполняют никаких действий). Если значение `count` оказалось больше `10`, управление передается первой инструкции, следующей за меткой `end:` (строка 20). В противном случае в строках 15–16 выводится текущее значение счетчика, затем оно увеличивается на 1, и управление передается первой инструкции, следующей за меткой `start:` (строка 9).

Пример 14.5 | Использование инструкции goto

```
1 // Пример 14.5: fig14_07.c
2 // Использование инструкции goto
3 #include <stdio.h>
4
5 int main( void )
6 {
7    int count = 1; // инициализировать count
8
9    start: // метка
10
11      if ( count > 10 ) {
12         goto end;
13      } // конец if
14
15      printf( "%d ", count );
16      ++count;
17
18      goto start; // перейти в строку 9
19
20   end: // метка
21      putchar( '\n' );
22 } // конец main
```

```
  1   2   3   4   5   6   7   8   9  10
```

В главе 3 отмечалось, что для написания программы любой сложности достаточно всего трех структур управления – последовательного выполнения, выбора и повторения. Следуя правилам структурного программирования, иногда можно прийти к созданию глубоко вложенных структур управления, откуда сложно выйти достаточно быстро. Некоторые программисты используют инструкции `goto` как инструмент быстрого выхода из глубоко вложенных структур управления. Это избавляет от необходимости выполнять массу дополнительных проверок. Кроме того, в иных ситуациях применение `goto` даже рекомендуется. Например, в рекомендации MEM12-C центра CERT говорится: «Подумайте о возможности использования цепочек переходов, когда из-за ошибки необходимо покинуть функцию и попутно освободить занятые ресурсы».

Инструкцию `goto` можно с успехом использовать для быстрого выхода из глубоко вложенных конструкций.

Применение инструкции `goto` идет вразрез с принципами структурного программирования и может осложнять отладку, сопровождение и дальнейшее развитие программ.

A

Таблица предшествования операторов

В табл. A.1 перечислены операторы в порядке уменьшения их приоритетов.

Таблица A.1 | Операторы языка C в порядке убывания приоритетов

Оператор	Тип	Ассоциативность
`()`	Круглые скобки (оператор вызова функции)	Слева направо
`[]`	Индекс в массиве	
`.`	Оператор доступа к члену объекта через сам объект	
`->`	Оператор доступа к члену объекта через указатель	
`++`	Унарный постинкремент	
`--`	Унарный постдекремент	
`++`	Унарный прединкремент	Справа налево
`--`	Унарный преддекремент	
`+`	Унарный плюс	
`-`	Унарный минус	
`!`	Унарный оператор логического отрицания	
`~`	Унарный оператор поразрядного дополнения	
(*тип*)	Оператор приведения к типу	
`*`	Разыменование	
`&`	Взятие адреса	
`sizeof`	Определение размера в байтах	

Таблица А.1 | (*окончание*)

Оператор	Тип	Ассоциативность
*	Умножение	Слева направо
/	Деление	
%	Деление по модулю (остаток от деления нацело)	
+	Сложение	Слева направо
-	Вычитание	
<<	Поразрядный сдвиг влево	Слева направо
>>	Поразрядный сдвиг вправо	
<	Оператор отношения «меньше, чем»	Слева направо
<=	Оператор отношения «меньше или равно»	
>	Оператор отношения «больше, чем»	
>=	Оператор отношения «больше или равно»	
==	Оператор отношения «равно»	Слева направо
!=	Оператор отношения «не равно»	
&	Поразрядное «И»	Слева направо
^	Поразрядное «ИСКЛЮЧАЮЩЕЕ ИЛИ»	Слева направо
|	Поразрядное «ИЛИ»	Слева направо
&&	Логическое «И»	Слева направо
||	Логическое «ИЛИ»	Слева направо
?:	Тернарный условный оператор	Справа налево
=	Присваивание	Справа налево
+=	Присваивание со сложением	
-=	Присваивание с вычитанием	
*=	Присваивание умножением	
/=	Присваивание с делением	
%=	Присваивание с делением по модулю (остатком от деления нацело)	
&=	Присваивание с поразрядным «И»	
^=	Присваивание с поразрядным «ИСКЛЮЧАЮЩЕЕ ИЛИ»	
|=	Присваивание с поразрядным «ИЛИ»	
<<=	Присваивание с поразрядным сдвигом влево	
>>=	Присваивание с поразрядным сдвигом вправо	
,	Запятая	Слева направо

B

Набор символов ASCII

Таблица B.1 | Набор символов ASCII

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	lf	vt	ff	cr	so	si	dle	dc1	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	«	#	$	%	&	‘
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	’	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

Цифры в левой колонке табл. B.1 соответствуют разрядам десятков и сотен в десятичном коде символов (0–127), а цифры в верхней строке табл. B.1 – разряду единиц в коде символов. Например, символ «F» имеет код 70, а символ «&» – код 38.

Системы счисления

В этом приложении вы:

- познакомитесь с основными понятиями систем счисления, такими как «основание», позиционное значение и символическое значение;
- узнаете, как работать с числами, представленными в двоичной, восьмеричной и шестнадцатеричной системах счисления;
- научитесь преобразовывать двоичные числа в восьмеричное или шестнадцатеричное представление;
- научитесь преобразовывать восьмеричные или шестнадцатеричные числа в двоичное представление;
- научитесь преобразовывать десятичные числа в двоичное, восьмеричное и шестнадцатеричное представление и обратно;
- познакомитесь с особенностями двоичной арифметики и узнаете, как можно представить отрицательное число с использованием нотации дополнения до двух.

С.1 Введение

В этом приложении рассматриваются ключевые системы счисления, используемые программистами, особенно при работе над программными проектами, требующими тесного взаимодействия с аппаратурой на низком уровне. К проектам такого рода относятся операционные системы, программы, взаимодействующие с сетью, компиляторы, системы управления базами данных и приложения, которые должны обеспечивать высочайшую производительность.

Добавляя в текст программы числа, такие как 227 или –63, предполагается, что они записываются в **десятичной системе счисления** (**по основанию 10**). Роль **цифр** в десятичной системе играют символы 0, 1, 2, 3, 4, 5, 6, 7, 8 и 9. Наименьшей цифрой является 0, а наибольшей – 9. Все они меньше **основания** 10. Внутри компьютеры используют **двоичную систему счисления** (**по основанию 2**). В двоичной системе счисления имеются всего две цифры, 0 и 1. Наименьшей цифрой является 0, а наибольшей – 1. Все они меньше основания 2.

Как будет показано ниже, двоичные числа обычно имеют более длинную форму записи, чем их десятичные эквиваленты. Программисты, работающие с языками Ассемблера и с высокоуровневыми языками, такими как C, позволяющими опускаться до машинного уровня, считают, что работать с двоичными числами неудобно. Поэтому особую популярность приобрели две другие системы счисления – **восьмеричная (по основанию 8)** и **шестнадцатеричная (по основанию 16)** – позволяющие записывать двоичные числа в более краткой форме.

В восьмеричной системе счисления используются цифры от 0 до 7. Так как обе системы счисления – двоичная и восьмеричная – имеют меньше цифр, чем десятичная, значения их цифр соответствуют значениям цифр в десятичной системе.

Шестнадцатеричная система счисления порождает определенную проблему, так как требует 16 цифр для представления чисел – наименьшая цифра

0, а наибольшая должна иметь значение, эквивалентное десятичному числу 15 (на единицу меньше основания 16). По общепринятому соглашению, для представления цифр с десятичными значениями от 10 до 15 используются буквы латинского алфавита от A до F. То есть шестнадцатеричные числа могут состоять исключительно из десятичных цифр, как, например, число 876, из цифр и букв, как число 8A55F, и только из букв, как число FFE. Иногда шестнадцатеричные числа могут напоминать слова английского языка, такие как FACE (лицо) или FEED (подача), что может показаться странным для программистов, привыкших работать только с десятичными числами. В табл. C.1 и C.2 перечислены цифры, используемые в двоичной, восьмеричной и шестнадцатеричной системах счисления.

В каждой из этих систем счисления используется **позиционная форма записи** – каждой позиции, в которой находится цифра, соответствует иное, **позиционное значение**. Например, в десятичном числе 937 (9, 3 и 7 называют символическими значениями) цифра 7 находится в разряде (в позиции) единиц, цифра 3 – в разряде десятков, а цифра 9 – в разряде сотен. Каждая из этих позиций является степенью основания (в данном случае степенью 10), причем отсчет степеней начинается с 0 и увеличивается на 1 в каждой следующей позиции, справа налево (табл. C.3).

Таблица C.1 | Цифры в двоичной, восьмеричной, десятичной и шестнадцатеричной системах счисления

Двоичные цифры	Восьмеричные цифры	Десятичные цифры	Шестнадцатеричные цифры
0	0	0	0
1	1	1	1
	2	2	2
	3	3	3
	4	4	4
	5	5	5
	6	6	6
	7	7	7
		8	8
		9	9
			A (десятичное значение 10)
			B (десятичное значение 11)
			C (десятичное значение 12)
			D (десятичное значение 13)
			E (десятичное значение 14)
			A (десятичное значение 15)

Таблица C.2 | Сравнительные характеристики двоичной, восьмеричной, десятичной и шестнадцатеричной систем счисления

Атрибут	Двоичная	Восьмеричная	Десятичная	Шестнадцатеричная
Основание	2	8	10	16
Наименьшая цифра	0	0	0	0
Наибольшая цифра	1	7	9	F

Таблица C.3 | Позиционные значения в десятичной системе счисления

Десятичная цифра	9	3	7
Имя позиции (разряда)	Сотни	Десятки	Единицы
Значение позиции	100	10	1
Значение позиции как степень основания (10)	10^2	10^1	10^0

В больших десятичных числах следующие разряды слева соответствуют тысячам (10 в степени 3), десяткам тысяч (10 в степени 4), сотням тысяч (10 в степени 5), миллионам (10 в степени 6), десяткам миллионов (10 в степени 7) и т. д.

В двоичном числе 101 самая правая цифра 1 находится в разряде единиц, цифра 0 – в разряде двоек и самая левая цифра 1 – в разряде четверок. Каждая из этих позиций является степенью основания (в данном случае 2), причем отсчет степеней начинается с 0 и увеличивается на 1 в каждой следующей позиции, справа налево (табл. C.4). То есть $101 = 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 4 + 0 + 1 = 5$.

Таблица C.4 | Позиционные значения в двоичной системе счисления

Двоичная цифра	1	0	1
Имя позиции (разряда)	Четверки	Двойки	Единицы
Значение позиции	4	2	1
Значение позиции как степень основания (2)	2^2	2^1	2^0

В больших двоичных числах следующие разряды слева соответствуют восьмеркам (2 в степени 3), двойкам в четвертой степени (16), двойкам в пятой степени (32), двойкам в шестой степени (64) и т. д.[1]

[1] В оригинале автор использует сложносоставные конструкции числительных, такие как «шестнадцатерки», «тридцатидвойки», «шестидесятичетверки», однако в русском языке это не принято, поэтому здесь и далее подобные «числительные» я заменил степенями. – *Прим. перев.*

В восьмеричном числе 425 цифра 5 находится в разряде единиц, цифра 2 – в разряде восьмерок и цифра 4 – в разряде восьмерок во второй степени. Каждая из этих позиций является степенью основания (в данном случае 8), причем отсчет степеней начинается с 0 и увеличивается на 1 в каждой следующей позиции, справа налево (табл. C.5).

Таблица C.5 | Позиционные значения в восьмеричной системе счисления

Восьмеричная цифра	4	2	5
Имя позиции (разряда)	Восьмерки во второй степени	Восьмерки	Единицы
Значение позиции	64	8	1
Значение позиции как степень основания (2)	8^2	8^1	8^0

В больших восьмеричных числах следующие разряды слева соответствуют восьмеркам в третьей степени (512), восьмеркам в четвертой степени (4096), восьмеркам в пятой степени (32768) и т. д.

В шестнадцатеричном числе 3DA цифра A находится в разряде единиц, цифра D – в разряде шестнадцати в первой степени и цифра 3 – в разряде шестнадцати во второй степени. Каждая из этих позиций является степенью основания (в данном случае 8), причем отсчет степеней начинается с 0 и увеличивается на 1 в каждой следующей позиции, справа налево (табл. C.6).

Таблица C.6 | Позиционные значения в шестнадцатеричной системе счисления

Шестнадцатеричная цифра	3	D	A
Имя позиции (разряда)	Шестнадцать во второй степени	Шестнадцать	Единицы
Значение позиции	256	16	1
Значение позиции как степень основания (2)	16^2	16^1	16^0

В больших шестнадцатеричных числах следующие разряды слева соответствуют шестнадцати в третьей степени (4096), шестнадцати в четвертой степени (65536) и т. д.

C.2 Преобразование двоичных чисел в восьмеричное и шестнадцатеричное представление

Восьмеричная и шестнадцатеричная формы записи в информатике используются в основном для более краткого представления двоичных чисел. Таб-

лица С.7 подчеркивает тот факт, что длинные двоичные числа могут быть представлены более компактно в системах счисления с большим основанием.

Таблица С.7 | Десятичные, двоичные, восьмеричные и шестнадцатеричные эквиваленты

Десятичное число	Двоичное представление	Восьмеричное представление	Шестнадцатеричное представление
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Особо следует подчеркнуть, что основаниями обеих систем счисления, восьмеричной и шестнадцатеричной, являются степени двойки (8 и 16 соответственно) – основания двоичной системы счисления. Взгляните на следующее 12-значное двоичное число и его восьмеричный и шестнадцатеричный эквиваленты. Сможете ли вы сами определить особенности соотношения этих чисел? Ответ на этот вопрос следует чуть ниже.

```
Двоичное число    Восьмеричный эквивалент    Шестнадцатеричный эквивалент
100011010001      4321                       8D1
```

Чтобы убедиться, насколько просто двоичные числа преобразуются в восьмеричные, просто разбейте 12-значное двоичное число на группы по три бита и запишите вместо групп соответствующие им восьмеричные цифры:

```
100  011  010  001
4    3    2    1
```

Восьмеричные цифры, записанные под группами из трех битов, являются восьмеричными эквивалентами соответствующих 3-значных двоичных чисел, как показано в табл. С.7.

Аналогичное соотношение наблюдается между двоичными и шестнадцатеричными числами. Разбейте 12-значное двоичное число на группы по четыре бита и запишите вместо групп соответствующие им шестнадцатеричные цифры:

```
1000  1101  0001
8     D     1
```

Шестнадцатеричные цифры, записанные под группами из четырех битов, являются шестнадцатеричными эквивалентами соответствующих 4-значных двоичных чисел, как показано в табл. C.7.

C.3 Преобразование восьмеричных и шестнадцатеричных чисел в двоичное представление

В предыдущем разделе было показано, как преобразовывать двоичные числа в восьмеричные и шестнадцатеричные эквиваленты, разбивая исходное число на группы цифр и просто заменяя их эквивалентными восьмеричными или шестнадцатеричными цифрами. Этот процесс можно обратить в противоположную сторону с целью преобразования восьмеричных и шестнадцатеричных чисел в двоичные эквиваленты.

Например, чтобы преобразовать восьмеричное число 653 в двоичное, достаточно заменить цифру 6 ее 3-значным двоичным эквивалентом 110, цифру 5 – 3-значным двоичным эквивалентом 101 и цифру 3 – 3-значным двоичным эквивалентом 011. В результате получится 9-значное двоичное число 110101011.

Чтобы преобразовать шестнадцатеричное число FAD5 в двоичное, достаточно заменить цифру F ее 4-значным двоичным эквивалентом 1111, цифру A – 4-значным двоичным эквивалентом 1010, цифру D – 4-значным двоичным эквивалентом 1101 и цифру 5 – 4-значным двоичным эквивалентом 0101. В результате получится 16-разрядное двоичное число 1111101011010101.

C.4 Преобразование двоичных, восьмеричных и шестнадцатеричных чисел в десятичное представление

Мы привыкли работать с десятичными числами, и поэтому часто бывает желательно преобразовать двоичное, восьмеричное или шестнадцатеричное число в десятичное, чтобы получить представление о величине числа. Таблицы в разделе C.1 описывают значения разрядов в разных системах счисления. Чтобы преобразовать число из любой другой системы счисления

в десятичную, нужно умножить десятичный эквивалент каждой цифры на значение позиции и сложить полученные произведения. Например, в табл. C.8 показано, как двоичное число 110101 преобразуется в десятичное 53.

Таблица C.8 | Преобразование двоичного числа в десятичное

Значения позиций	32	16	8	4	2	1
Символические значения	1	1	0	1	0	1
Произведения	1*32 = 32	1*16 = 16	0*8 = 0	1*4 = 4	0*2 = 0	1*1 = 1
Сумма	= 32 + 16 + 0 + 4 + 0 + 1 = 53					

Преобразование восьмеричного числа 7614 в десятичное 3980 выполняется аналогично, только на этот раз используются значения позиций, соответствующие восьмеричной системе счисления, как показано в табл. C.9.

Таблица C.9 | Преобразование восьмеричного числа в десятичное

Значения позиций	512	64	8	1
Символические значения	7	6	1	4
Произведения	7*512 = 3584	6*64 = 384	1*8 = 8	4*1 = 4
Сумма	= 3584 + 384 + 8 + 4 = 3980			

Преобразование шестнадцатеричного числа AD3B в десятичное 44347 выполняется точно так же, но с использованием значений позиций, соответствующих шестнадцатеричной системе счисления, как показано в табл. C.10.

Таблица C.10 | Преобразование шестнадцатеричного числа в десятичное

Значения позиций	4096	256	16	1
Символические значения	A	D	3	B
Произведения	A*4096 = 40960	D*256 = 3328	3*16 = 48	B*1 = 11
Сумма	= 40960 + 3328 + 48 + 11 = 44347			

C.5 Преобразование десятичных чисел в двоичное, восьмеричное и шестнадцатеричное представление

Преобразования, описанные в разделе C.4, вытекают из позиционной природы обсуждаемых систем счисления. Преобразование десятичных чисел в двоичную, восьмеричную или шестнадцатеричную систему счисления также вытекает из позиционной природы.

Допустим, что требуется преобразовать десятичное число 57 в двоичное. Для начала добавим в таблицу преобразования колонки справа налево, с позиционными значениями, пока не будет достигнуто позиционное значение, превышающее исходное десятичное число. Эта колонка нам не потребуется, поэтому просто удалим ее. То есть сначала у нас получится таблица:

```
Значения позиций: 64   32   16    8    4    2    1
```

а затем после удаления столбца с позиционным значением 64:

```
Значения позиций: 32   16    8    4    2    1
```

Далее начинаем движение по столбцам слева направо. Делим 57 на 32, получаем частное 1 и остаток 25, записываем 1 в колонку с заголовком 32. Делим 25 на 16, получаем частное 1 и остаток 9, записываем 1 в колонку с заголовком 16. Делим 9 на 8, получаем частное 1 и остаток 1. В следующих двух колонках частное получается равным 0, так как соответствующие им позиционные значения больше делимого 1, поэтому записываем 0 в колонки с заголовками 4 и 2. Наконец, делим 1 на 1, получаем частное 1 и записываем 1 в колонку с заголовком 1:

```
Значения позиций:       32   16    8    4    2    1
Символические значения: 1    1     1    0    0    1
```

То есть десятичное число 57 в двоичной системе имеет вид 111001.

Теперь преобразуем десятичное число 103 в восьмеричное. Для начала добавим в таблицу преобразования колонки справа налево, пока не будет достигнуто позиционное значение, превышающее исходное десятичное число. Эта колонка нам не потребуется, поэтому просто удалим ее. То есть сначала у нас получится таблица:

```
Значения позиций: 512   64   8    1
```

а затем после удаления столбца с позиционным значением 512:

```
Значения позиций: 64   8    1
```

Далее начинаем движение по столбцам слева направо. Делим 103 на 64, получаем частное 1 и остаток 39, записываем 1 в колонку с заголовком 64. Делим 39 на 8, получаем частное 4 и остаток 7, записываем 1 в колонку с заголовком 8. Наконец, делим 7 на 1, получаем частное 7 и остаток 0, поэтому записываем 7 в колонку с заголовком 1:

```
Значения позиций:       64   8    1
Символические значения: 1    4    7
```

То есть десятичное число 103 в восьмеричной системе имеет вид 147.

Теперь преобразуем десятичное число 375 в шестнадцатеричное. Для начала добавим в таблицу преобразования колонки справа налево, пока не будет достигнуто позиционное значение, превышающее исходное десятич-

ное число. Эта колонка нам не потребуется, поэтому просто удалим ее. То есть сначала у нас получится таблица:

```
Значения позиций: 4096   256   16    1
```

а затем после удаления столбца с позиционным значением 4096:

```
Значения позиций: 256   16    1
```

Далее начинаем движение по столбцам слева направо. Делим 375 на 256, получаем частное 1 и остаток 119, записываем 1 в колонку с заголовком 256. Делим 119 на 16, получаем частное 7 и остаток 7, записываем 7 в колонку с заголовком 16. Наконец, делим 7 на 1, получаем частное 7 и остаток 0, поэтому записываем 7 в колонку с заголовком 1:

```
Значения позиций:       256   16    1
Символические значения: 1     7     7
```

То есть десятичное число 375 в шестнадцатеричной системе имеет вид: 177.

С.6 Отрицательные двоичные числа: нотация дополнения до двух

До сих пор мы рассматривали только положительные числа. В этом разделе мы посмотрим, как компьютеры представляют отрицательные числа, с использованием **нотации дополнения до двух**. Сначала посмотрим, как выполняется дополнение до двух для двоичных чисел, а затем узнаем, почему оно считается отрицательным значением данного двоичного числа.

В качестве примера возьмем 32-битное целое со знаком:

```
int value = 13;
```

В 32-разрядном представлении переменная `value` имеет следующее значение:

```
00000000 00000000 00000000 00001101
```

Чтобы получить отрицательное значение **value**, сначала следует найти дополнение до единицы, применив **поразрядный оператор дополнения** (~):

```
onesComplementOfValue = ~value;
```

Во внутреннем представлении значение `~value` соответствует значению `value`, в котором значения всех битов были изменены на противоположные – нули стали единицами, а единицы стали нулями, как показано ниже:

```
value:
00000000 00000000 00000000 00001101

~value (то есть дополнение value до единицы):
11111111 11111111 11111111 11110010
```

Чтобы получить дополнение значения `value` до двух, нужно просто прибавить 1 к значению дополнения `value` до единицы, то есть

```
Дополнение значения value до двух:
11111111 11111111 11111111 11110011
```

Теперь, если это число равно –13, при сложении с числом 13 мы должны получить 0. Проверим:

```
 00000000 00000000 00000000 00001101
+11111111 11111111 11111111 11110011
-----------------------------------
 00000000 00000000 00000000 00000000
```

Бит переноса оказывается левее крайнего левого разряда и отбрасывается, и мы действительно получаем 0. Если сложить число с дополнением этого числа до единицы, мы получим 1. Причина получения значения 0 в результате состоит в том, что дополнение до двух на единицу больше дополнения до единицы. Прибавление 1 обусловливает необходимость сложения каждого разряда с 0 и с битом переноса 1. Бит переноса продолжает смещаться влево, пока не выйдет за крайний левый разряд, в результате чего мы получаем 0 во всех битах.

Фактически операцию вычитания

```
x = a - value;
```

компьютеры выполняют как сложение числа `a` с дополнением числа `value` до двух:

```
x = a + (~value + 1);
```

Допустим, что `a` имеет значение 27, а `value` – значение 13. Если дополнение значения `value` до двух действительно является отрицательным значением с той же абсолютной величиной, тогда сложение значения `a` с дополнением значения `value` до двух должно вернуть число 14. Проверим:

```
a (то есть 27) 00000000 00000000 00000000 00011011
+(~value + 1)  +11111111 11111111 11111111 11110011
               -----------------------------------
                00000000 00000000 00000000 00001110
```

Результат действительно равен 14.

Сортировка: взгляд в глубину

В этом приложении вы познакомитесь:

- с сортировкой массивов с использованием алгоритма сортировки методом выбора;
- с сортировкой массивов с применением алгоритма сортировки методом вставки;
- с сортировкой массивов с использованием рекурсивного алгоритма сортировки методом слияния;
- с определением эффективности алгоритмов поиска и сортировки и научитесь выражать ее в нотации «Большое О».

Краткое содержание

D.1 Введение

Как уже рассказывалось в главе 6, сортировка данных выполняется в порядке возрастания или убывания по одному или нескольким ключам сортировки. В этом приложении рассказывается об алгоритмах сортировки методом выбора и методом вставки, а также о наиболее эффективном, но более сложном алгоритме сортировки методом слияния. Мы познакомимся с **нотацией «Большое О»**, используемой для оценки времени работы алгоритма в худшем случае, то есть для оценки объема работы, который придется выполнить алгоритму, чтобы решить поставленную задачу.

Важно понимать, что независимо от выбранного алгоритма должен получаться один и тот же результат – отсортированный массив с данными. Выбор алгоритма влияет только на время выполнения сортировки и на потребление памяти. Первые два алгоритма сортировки, которые будут представлены здесь, – сортировка методом выбора и сортировка методом вставки – очень просты в реализации, но крайне неэффективны. Третий алгоритм – рекурсивный алгоритм сортировки методом слияния – гораздо эффективнее алгоритмов, упомянутых выше, но он очень сложен в реализации.

D.2 Нотация «Большое О»

Допустим, что имеется алгоритм, проверяющий равенство первого и второго элементов массива. Если предположить, что массив имеет 10 элементов, этому алгоритму потребуется выполнить всего одно сравнение. Если массив будет иметь 1000 элементов, алгоритму все равно потребуется выполнить только одно сравнение. Фактически данный алгоритм полностью независим от количества элементов в массиве. Про такой алгоритм говорят, что он имеет **постоянное время выполнения**, которое в нотации «Большое О» обозначается как ***O*(1)**. Алгоритм со сложностью *O*(1) не обязательно будет выполнять единственное сравнение. Запись *O*(1) означает лишь, что количество сравнений остается *постоянным* – оно не зависит от размеров массива. Алгоритм, сравнивающий первый элемент массива с последующими тремя, также будет иметь сложность *O*(1), даже при том, что выполняет три сравнения.

Алгоритм, сравнивающий первый элемент массива с остальными элементами *до первого совпадения*, должен выполнить до $n - 1$ сравнений, где n – количество элементов в массиве. Если массив содержит 10 элементов, алгоритму потребуется выполнить до девяти сравнений. Если массив содержит 1000 элементов, алгоритму потребуется выполнить до 999 сравнений. С ростом количества элементов часть n выражения будет становиться

все более «весомой» и уменьшение на единицу будет оказывать все менее существенное влияние. Нотация «Большое О» обеспечивает выделение «весомых» факторов и игнорирует те, значимость которых падает с ростом n. По этой причине про алгоритм, выполняющий до $n - 1$ сравнений (как, например, описанный выше), говорят, что он имеет сложность $O(n)$. Алгоритм со сложностью $O(n)$ называют алгоритмом с линейной сложностью.

Представим, что имеется алгоритм, выполняющий поиск дубликатов *любого* элемента в массиве. Он должен сравнить первый элемент со всеми остальными элементами в массиве. Второй элемент – со всеми остальными, кроме первого: сравнение первого и второго элементов уже было выполнено при поиске дубликатов первого элемента. Третий элемент – со всеми остальными, кроме первых двух. В конечном итоге этому алгоритму потребуется выполнить $(n - 1) + (n - 2) + ... + 2 + 1$, или $n^2/2 - n/2$, сравнений. С ростом n увеличивается вес фактора n^2, а вес фактора n уменьшается. Нотация «Большое О» подчеркивает рост весомости фактора n^2, отбрасывая $n^2/2$. И, как будет показано далее, постоянные факторы также отбрасываются в нотации «Большое О».

Главная задача нотации «Большое О» – показать, как растет время выполнения алгоритма с увеличением количества обрабатываемых элементов. Допустим, что алгоритму требуется выполнить n^2 сравнений. Для массива с четырьмя элементами алгоритм выполнит 16 сравнений; для массива с восемью элементами – 64 сравнения. Удвоение количества элементов приводит к увеличению сравнений, выполняемых таким алгоритмом, в четыре раза. Рассмотрим аналогичный алгоритм, требующий выполнить $n^2/2$ сравнений. Для массива с четырьмя элементами алгоритм выполнит восемь сравнений; для массива с восемью элементами – 32 сравнения. Опять же, удвоение количества элементов приводит к увеличению сравнений в четыре раза. Время выполнения обоих алгоритмов растет как квадрат количества элементов n, поэтому нотация «Большое О» игнорирует константу и оба алгоритма обозначаются как $\boldsymbol{O(n^2)}$ – про такие алгоритмы говорят, что они имеют квадратичную сложность.

Для маленьких значений n алгоритмы со сложностью $O(n^2)$ (на современных компьютерах, способных выполнять миллиарды операций в секунду) не оказывают существенного влияния на производительность. Но с ростом n падение производительности будет проявляться все сильнее и сильнее. Для обработки массива с миллионом элементов алгоритму со сложностью $O(n^2)$ требуется выполнить триллион «операций» (каждая из которых может в действительности требовать выполнения множества машинных инструкций). Обработка массива с применением такого алгоритма может занимать несколько часов. Для обработки массива с миллиардом элементов потребуется выполнить квинтильон операций. Это число настолько велико, что для решения поставленной задачи может понадобиться несколько десятилетий! К сожалению, алгоритмы со сложностью $O(n^2)$ легко реализуются, как будет показано далее в этом приложении. Здесь вы также увидите алгоритм с более выгодным значением «Большого О». Часто для создания более

эффективного алгоритма приходится прикладывать значительные усилия и проявлять смекалку, но преимущества производительности стоят дополнительных затрат, особенно при больших значениях *n*.

D.3 Сортировка методом выбора

Алгоритм **сортировки методом выбора** прост, но неэффективен. В первой итерации алгоритма выбирается наименьший элемент в массиве и меняется местами с первым элементом. Во второй итерации выбирается второй наименьший элемент (из всех элементов, кроме первого) и меняется местами со вторым элементом. Алгоритм продолжает выполняться, пока в последней итерации программа не выберет второй по величине элемент и не поменяет его местами с элементом, вторым с конца, в результате чего самый большой элемент окажется последним. После *i*-й итерации окажутся отсортированы первые *i* элементов в массиве.

Возьмем в качестве примера следующий массив:

```
34  56   4  10  77  51  93  30   5  52
```

Программа, реализующая алгоритм сортировки методом выбора, сначала найдет в этом массиве наименьший элемент (третий элемент со значением 4, то есть элемент с индексом 2, потому что нумерация индексов в массивах начинается с 0) и поменяет местами элементы со значениями 4 и 34, в результате чего получится массив:

```
4  56  34  10  77  51  93  30   5  52
```

Затем программа определит второй наименьший элемент (среди всех элементов, кроме первого) – элемент с индексом 8 и значением 5 – и поменяет местами элементы со значениями 5 и 56, в результате чего получится массив:

```
4   5  34  10  77  51  93  30  56  52
```

В третьей итерации программа поменяет местами элементы со значениями 10 и 34.

Процесс будет продолжаться, пока в результате выполнения девяти итераций массив не будет полностью отсортирован.

```
4   5  10  30  34  51  52  56  77  93
```

После первой итерации наименьший элемент займет первую позицию в массиве. После второй итерации два наименьших элемента займут две первые позиции. После третьей итерации три наименьших элемента займут три первые позиции.

В примере D.1 представлена реализация сортировки методом выбора массива `array`, инициализированного десятью случайными числами (возможно, повторяющимися). Функция `main` выводит несортированный массив, вызывает функцию `sort` и затем выводит массив еще раз, после его сортировки.

Пример D.1 | Алгоритм сортировки методом выбора

```
/* Пример D.1: figD_01.c
   Алгоритм сортировки методом выбора. */
#define SIZE 10
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/* прототипы функций */
void selectionSort( int array[], int length );
void swap( int array[], int first, int second );
void printPass( int array[], int length, int pass, int index );

int main( void )
{
   int array[ SIZE ]; /* объявление массива целых чисел для сортировки */
   int i;             /* для использования в цикле for */

   srand( time( NULL ) ); /* установить начальное значение для rand */

   for ( i = 0; i < SIZE; i++ )
      array[ i ] = rand() % 90 + 10; /* инициализировать элементы */

   puts( "Unsorted array:" );

   for ( i = 0; i < SIZE; i++ )      /* вывести массив */
      printf( "%d ", array[ i ] );

   puts( "\n" );
   selectionSort( array, SIZE );
   puts( "Sorted array:" );

   for ( i = 0; i < SIZE; i++ )      /* вывести массив */
      printf( "%d ", array[ i ] );

} /* конец функции main */

/* функция сортировки массива методом выбора */
void selectionSort( int array[], int length )
{
   int smallest; /* индекс наименьшего элемента */
   int i, j;     /* используются в циклах for */

   /* цикл по первым length - 1 элементам */
   for ( i = 0; i < length - 1; i++ ) {
      smallest = i; /* первый индекс неотсортированной части массива */

      /* цикл поиска наименьшего элемента */
      for ( j = i + 1; j < length; j++ )
         if ( array[ j ] < array[ smallest ] )
            smallest = j;

      swap( array, i, smallest );                /* поменять местами */
      printPass( array, length, i + 1, smallest ); /* результат итерации */
   } /* конец for */
} /* конец функции selectionSort */

/* функция, меняющая местами два элемента в массиве */
void swap( int array[], int first, int second )
{
   int temp; /* временная переменная */
   temp = array[ first ];
   array[ first ] = array[ second ];
```

```
63    array[ second ] = temp;
64 } /* конец функции swap */
65
66 /* функция вывода результатов итерации */
67 void printPass( int array[], int length, int pass, int index )
68 {
69    int i; /* используется в цикле for */
70
71    printf( "After pass %2d: ", pass );
72
73    /* вывести элементы до выбранного */
74    for ( i = 0; i < index; i++ )
75       printf( "%d ", array[ i ] );
76
77    printf( "%d* ", array[ index ] ); /* отметить элемент, */
78                                      /* с которым произошел обмен */
79    /* вывести остаток массива */
80    for ( i = index + 1; i < length; i++ )
81       printf( "%d ", array[ i ] );
82
83    printf( "%s", "\n              " ); /* для выравнивания */
84
85    /* вывести индикатор хода сортировки массива */
86    for ( i = 0; i < pass; i++ )
87       printf( "%s", "--  " );
88
89    puts( "\n" ); /* добавить перевод строки*/
90 } /* конец функции printPass */
```

```
Unsorted array:
72  34  88  14  32  12  34  77  56  83
After pass  1: 12  34  88  14  32  72* 34  77  56  83
               --
After pass  2: 12  14  88  34* 32  72  34  77  56  83
               --  --
After pass  3: 12  14  32  34  88* 72  34  77  56  83
               --  --  --
After pass  4: 12  14  32  34* 88  72  34  77  56  83
               --  --  --  --
After pass  5: 12  14  32  34  34  72  88* 77  56  83
               --  --  --  --  --
After pass  6: 12  14  32  34  34  56  88  77  72* 83
               --  --  --  --  --  --
After pass  7: 12  14  32  34  34  56  72  77  88* 83
               --  --  --  --  --  --  --
After pass  8: 12  14  32  34  34  56  72  77* 88  83
               --  --  --  --  --  --  --  --
After pass  9: 12  14  32  34  34  56  72  77  83  88*
               --  --  --  --  --  --  --  --  --
After pass 10: 12  14  32  34  34  56  72  77  83  88*
               --  --  --  --  --  --  --  --  --  --
Sorted array:
12  14  32  34  34  56  72  77  83  88
```

Строки 38–55 определяют функцию `selectionSort`. В строке 40 объявляется переменная `smallest`, где будет храниться индекс наименьшего элемента в оставшейся части массива. В строках 44–54 выполняется цикл `SIZE - 1` раз. В строке 45 запоминается текущий индекс наименьшего элемента. В строках 48–50 выполняется цикл по оставшимся элементам в массиве. В строке 49 каждый из этих элементов сравнивается с наименьшим. Если текущий элемент меньше наименьшего, в строке 50 запоминается индекс

вновь найденного наименьшего элемента. Когда цикл завершится, `smallest` будет хранить индекс наименьшего элемента в оставшейся части массива. В строке 52 вызывается функция `swap` (строки 58–64), помещающая наименьший элемент в нужную позицию.

В выводе этой программы используются дефисы, отмечающие часть массива, гарантированно отсортированную после каждой итерации. Звездочка выводится за позицией, где был найден наименьший элемент в данной итерации. В каждой итерации выполняется обмен местами элементов слева от звездочки и над правой крайней парой дефисов.

Эффективность сортировки методом выбора

Алгоритм сортировки методом выбора имеет сложность $O(n^2)$. Функция `selectionSort`, объявленная в строках 38–55 в примере D.1, реализующая алгоритм сортировки методом выбора, содержит два цикла `for`. Внешний цикл `for` (строки 44–54) выполняет итерации по $n - 1$ первым элементам массива и перемещает наименьший элемент в требуемую позицию. Внутренний цикл `for` (строки 48–50) выполняет итерации по оставшимся элементам массива, отыскивая наименьший элемент. Этот цикл выполняется $n - 1$ раз в первой итерации внешнего цикла, $n - 2$ раз во второй итерации, затем $n - 3$, ..., 3, 2, 1. Всего внутренний цикл выполняется $n(n - 1) / 2$, или $(n^2 - n)/2$, раз. В нотации «Большого О» наименее значимые факторы отбрасываются, а константы игнорируются. В результате получается, что данный алгоритм имеет сложность $O(n^2)$.

D.4 Сортировка методом вставки

Алгоритм **сортировки методом вставки** – еще один простой, но неэффективный алгоритм сортировки. В первой итерации этого алгоритма второй элемент массива сравнивается с первым, и если он меньше, первый и второй элементы меняются местами. Во второй итерации исследуется третий элемент и вставляется в нужную позицию, с учетом первых двух элементов, то есть после второй итерации первые три элемента будут отсортированы. После i-й итерации будут отсортированы первые i элементов оригинального массива.

Рассмотрим в качестве примера следующий массив. [*Обратите внимание*: это тот же массив, что использовался в описании алгоритма сортировки методом выбора.]

```
34  56   4  10  77  51  93  30   5  52
```

Программа, реализующая алгоритм сортировки методом вставки, сначала сравнит первые два элемента массива со значениями 34 и 56. Эти два элемента уже стоят в правильном порядке, поэтому программа перейдет к следующей итерации (если бы они стояли в неправильном порядке, программа поменяла бы их местами).

В следующей итерации программа начнет исследовать третий элемент со значением 4. Это значение меньше 56, поэтому программа сохранит 4 во временной переменной и переместит элемент 56 на одну позицию вправо. Затем программа выполнит сравнение с первым элементом и обнаружит, что 4 меньше 34, поэтому она переместит 34 на одну позицию вправо. Так как после этого будет достигнуто начало массива, программа сохранит 4 в элементе с индексом 0. В результате чего получится массив:

```
4  10  34  56  77  51  93  30   5  52
```

После *i*-й итерации этого алгоритма будут отсортированы первые $i + 1$ элементов оригинального массива (относительно друг друга). Промежуточные результаты сортировки первых элементов могут оказаться неокончательными, потому что правее в массиве могут оказаться элементы с меньшими значениями.

В примере D.2 приводится реализация алгоритма сортировки методом вставки. В строках 37–57 определяется функция `insertionSort`. В строке 39 объявляется переменная `insert`, где будет храниться значение элемента для вставки, пока программа перемещает другие элементы вправо. В строках 43–56 выполняется цикл по `SIZE - 1` элементам массива. В каждой итерации в строке 45 выполняется сохранение значения элемента для вставки в отсортированную часть массива. В строке 44 объявляется и инициализируется переменная `moveItem`, в которой хранится позиция для вставки. Цикл в строках 48–52 отыскивает правильную позицию для вставки. Этот цикл завершается либо по достижении начала массива, либо при обнаружении элемента, значение которого меньше значения вставляемого элемента. Строка 50 перемещает элемент вправо, а строка 51 уменьшает номер позиции, куда будет вставлен следующий элемент. По завершении цикла строка 54 вставляет элемент на место. В выводе этой программы используются дефисы, отмечающие отсортированную часть массива после каждой итерации. Звездочка выводится за позицией, куда в данной итерации был вставлен элемент.

Пример D.2 | Алгоритм сортировки методом вставки

```
 1 /* Пример D.2: figD_02.c
 2    Алгоритм сортировки методом вставки. */
 3 #define SIZE 10
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <time.h>
 7
 8 /* прототипы функций */
 9 void insertionSort( int array[], int length );
10 void printPass( int array[], int length, int pass, int index );
11
12 int main( void )
13 {
14    int array[ SIZE ]; /* объявление массива целых чисел для сортировки */
15    int i;             /* для использования в цикле for */
16
```

```
17    srand( time( NULL ) ); /* установить начальное значение для rand */
18
19    for ( i = 0; i < SIZE; i++ )
20       array[ i ] = rand() % 90 + 10; /* инициализировать элементы */
21
22    puts( "Unsorted array:" );
23
24    for ( i = 0; i < SIZE; i++ )      /* вывести массив */
25       printf( "%d ", array[ i ] );
26
27    puts( "\n" );
28    insertionSort( array, SIZE );
29    puts( "Sorted array:" );
30
31    for ( i = 0; i < SIZE; i++ )      /* вывести массив */
32       printf( "%d ", array[ i ] );
33
34 } /* конец функции main */
35
36 /* функция сортировки массива */
37 void insertionSort( int array[], int length )
38 {
39    int insert; /* временная переменная для хранения вставляемого элемента */
40    int i;      /* используются в цикле for */
41
42    /* цикл по первым length - 1 элементам */
43    for ( i = 1; i < length; i++ ) {
44       int moveItem = i; /* инициализировать индекс для вставки элемента */
45       insert = array[ i ];
46
47       /* найти место для текущего элемента */
48       while ( moveItem > 0 && array[ moveItem - 1 ] > insert ) {
49          /* сдвинуть элемент вправо на 1 позицию */
50          array[ moveItem ] = array[ moveItem - 1 ];
51          --moveItem;
52       } /* конец while */
53
54       array[ moveItem ] = insert; /* вставить элемент */
55       printPass( array, length, i, moveItem );
56    } /* конец for */
57 } /* конец функции insertionSort */
58
59 /* функция вывода результатов итерации */
60 void printPass( int array[], int length, int pass, int index )
61 {
62    int i; /* используется в цикле for */
63
64    printf( "After pass %2d: ", pass );
65
66    /* вывести элементы до выбранного */
67    for ( i = 0; i < index; i++ )
68       printf( "%d ", array[ i ] );
69
70    printf( "%d* ", array[ index ] ); /* отметить элемент */
71                                      /* с которым произошел обмен */
72    /* вывести остаток массива */
73    for ( i = index + 1; i < length; i++ )
74       printf( "%d ", array[ i ] );
75
76    puts( "%s* ", "\n " );            /* для выравнивания */
77
78    /* вывести индикатор хода сортировки массива */
79    for ( i = 0; i <= pass; i++ )
```

```
80       printf( "%d* ", "--  " );
81
82    puts( "" ); /* добавить перевод строки */
83 } /* конец функции printPass */
```

```
Unsorted array:
72  16  11  92  63  99  59  82  99  30

After pass 1: 16* 72  11  92  63  99  59  82  99  30
               --  --
After pass 2: 11* 16  72  92  63  99  59  82  99  30
               --  --  --
After pass 3: 11  16  72  92* 63  99  59  82  99  30
               --  --  --  --
After pass 4: 11  16  63* 72  92  99  59  82  99  30
               --  --  --  --  --
After pass 5: 11  16  63  72  92  99* 59  82  99  30
               --  --  --  --  --  --
After pass 6: 11  16  59* 63  72  92  99  82  99  30
               --  --  --  --  --  --  --
After pass 7: 11  16  59  63  72  82* 92  99  99  30
               --  --  --  --  --  --  --  --
After pass 8: 11  16  59  63  72  82  92  99  99* 30
               --  --  --  --  --  --  --  --  --
After pass 9: 11  16  30* 59  63  72  82  92  99  99
               --  --  --  --  --  --  --  --  --  --
Sorted array:
11  16  30  59  63  72  82  92  99  99
```

Эффективность сортировки методом вставки

Алгоритм сортировки методом вставки также имеет сложность $O(n^2)$. Как и в реализации сортировки методом выбора, функция `insertionSort` (строки 37–57) содержит два цикла. Внешний цикл `for` (строки 43–56) выполняет итерации по `SIZE – 1` первым элементам массива и вставляет элементы в соответствующие позиции. В данном приложении выражение `SIZE - 1` эквивалентно выражению $n - 1$ (так как константа `SIZE` определяет размер массива). Внутренний цикл `while` (строки 48–52) выполняет итерации по предыдущим элементам массива. В худшем случае этому циклу потребуется выполнить $n - 1$ сравнений. Каждый отдельный цикл имеет сложность $O(n)$. В нотации «Большого О» наличие вложенного цикла означает, что количество итераций внешнего цикла необходимо умножить на количество итераций внутреннего цикла. В каждой итерации внешнего цикла будет выполняться определенное количество итераций внутреннего цикла. В данном алгоритме на каждую итерацию со сложностью $O(n)$ внешнего цикла приходится $O(n)$ итераций внутреннего цикла. Умножая значения, получаем $O(n^2)$.

D.5 Сортировка методом слияния

Сортировка методом слияния – весьма эффективный алгоритм, но концептуально более сложный, чем алгоритмы сортировки методом выбора

и методом вставки. Суть алгоритма сортировки методом слияния состоит в следующем: он разбивает массив на две равные части, сортирует каждую из частей, а затем объединяет их в один большой массив. Если исходный массив содержит нечетное количество элементов, алгоритм делит его так, что в одной части оказывается на один элемент больше, чем в другой.

В данном примере демонстрируется рекурсивная реализация алгоритма. Базовым случаем рекурсии является массив с одним элементом. Естественно, массив с единственным элементом по определению является отсортированным, поэтому функция сортировки немедленно возвращает управление. На каждом шаге рекурсии массив, состоящий из двух или более элементов, разбивается на две части разного размера, затем каждая из частей рекурсивно сортируется, после чего обе части объединяются в один отсортированный массив. [Опять же, если массив содержит нечетное количество элементов, алгоритм делит его так, что в одной части оказывается на один элемент больше, чем в другой.]

Допустим, что на некотором этапе работы алгоритма мы получили два отсортированных массива – массив A:

```
4 10 34 56 77
```

и массив B:

```
5 30 51 52 93
```

которые теперь требуется объединить в один большой массив. Наименьший элемент в массиве A (с индексом 0) имеет значение 4. Наименьший элемент в массиве B (с индексом 0) имеет значение 5. Чтобы определить наименьший элемент в объединенном массиве, алгоритм сравнивает 4 и 5. Значение элемента из массива A оказывается меньше, поэтому 4 становится первым элементом в объединенном массиве. Далее алгоритм сравнивает 10 (второй элемент в массиве A) с элементом 5 (первым элементом в массиве B). Значение элемента из массива B оказывается меньше, поэтому 5 становится вторым элементом в объединенном массиве. Далее сравниваются элементы 10 и 30, и элемент 10 становится третьим элементом объединенного массива, и т. д.

В примере D.3 представлена реализация алгоритма сортировки методом слияния, где в строках 34–37 определяется функция `mergeSort`. В строке 36 вызывается функция `sortSubArray` со значениями аргументов `0` и `SIZE - 1`. Аргументы соответствуют начальному и конечному индексам сортируемого массива, в результате чего `sortSubArray` выполняет сортировку всего массива. Функция `sortSubArray` определена в строках 40–65. В строке 45 проверяется базовый случай рекурсии. Если размер массива оказывается равным 1, считается, что массив уже отсортирован, поэтому функция просто возвращает управление. Если размер массива больше 1, функция разбивает этот массив на две части, рекурсивно вызывает `sortSubArray`, чтобы отсортировать их, и затем объединяет отсортированные части массива. В строке 59 выполняется рекурсивный вызов `sortSubArray` для первой половины мас-

сива, а в строке 60 – для второй половины. Когда оба вызова вернут управление, каждая половина массива будет отсортирована. В строке 63 вызывается функция `merge` (определена в строках 68–110) с двумя половинами массива, которая объединяет их в один отсортированный массив.

Пример D.3 | Алгоритм сортировки методом слияния

```
 1 /* Пример D.3: figD_03.c
 2    Алгоритм сортировки методом слияния. */
 3 #define SIZE 10
 4 #include <stdio.h>
 5 #include <stdlib.h>
 6 #include <time.h>
 7
 8 /* прототипы функций */
 9 void mergeSort( int array[], int length );
10 void sortSubArray( int array[], int low, int high );
11 void merge( int array[], int left, int middle1, int middle2, int right );
12 void displayElements( int array[], int length );
13 void displaySubArray( int array[], int left, int right );
14
15 int main( void )
16 {
17    int array[ SIZE ]; /* объявление массива целых чисел для сортировки */
18    int i;             /* для использования в цикле for */
19
20    srand( time( NULL ) ); /* установить начальное значение для rand */
21
22    for ( i = 0; i < SIZE; i++ )
23       array[ i ] = rand() % 90 + 10; /* инициализировать элементы */
24
25    puts( "Unsorted array:" );
26    displayElements( array, SIZE );   /* вывести массив */
27    puts( "\n" );
28    mergeSort( array, SIZE );         /* отсортировать массив */
29    puts( "Sorted array:" );
30    displayElements( array, SIZE );   /* вывести массив */
31 } /* конец фунукции main */
32
33 /* функция сортировки методом слияния */
34 void mergeSort( int array[], int length )
35 {
36    sortSubArray( array, 0, length - 1 );
37 } /* конец функции mergeSort */
38
39 /* функция сортировки половинки массива */
40 void sortSubArray( int array[], int low, int high )
41 {
42    int middle1, middle2; /* хранят индексы, по которым разбит массив */
43
44    /* проверить базовый случай: размер массива равен 1 */
45    if ( ( high - low ) >= 1 ) { /* если не базовый случай... */
46       middle1 = ( low + high ) / 2;
47       middle2 = middle1 + 1;
48
49       /* вывести результат разбиения */
50       printf( "%s* ", "split:  " );
51       displaySubArray( array, low, high );
52       printf( "%s* ", "\n         " );
53       displaySubArray( array, low, middle1 );
54       printf( "%s* ", "\n           " );
55       displaySubArray( array, middle2, high );
```

```
56       puts( "\n" );
57
58       /* отсортировать половинки массива */
59       sortSubArray( array, low, middle1 );  /* первая половина */
60       sortSubArray( array, middle2, high ); /* вторая половина */
61
62       /* объединить отсортированные половинки */
63       merge( array, low, middle1, middle2, high );
64    } /* конец if */
65 } /* конец функции sortSubArray */
66
67 /* объединяет два отсортированных массива в один */
68 void merge( int array[], int left, int middle1, int middle2, int right )
69 {
70    int leftIndex = left;     /* индекс в левой половине массива */
71    int rightIndex = middle2; /* индекс в правой половине массива */
72    int combinedIndex = left; /* индекс во временном массиве */
73    int tempArray[ SIZE ];    /* временный массив */
74    int i;                    /* для использования в цикле for */
75
76    /* вывести два массива перед объединением */
77    printf( "%s* ", "merge:   " );
78    displaySubArray( array, left, middle1 );
79    printf( "%s* ", "\n         " );
80    displaySubArray( array, middle2, right );
81    puts( "" );
82
83    /* объединять массивы, пока не будет достигнут конец одного из них */
84    while ( leftIndex <= middle1 && rightIndex <= right ) {
85       /* поместить меньший из элементов в результат */
86       /* и перейти к следующей позиции в половине массива */
87       if ( array[ leftIndex ] <= array[ rightIndex ] )
88          tempArray[ combinedIndex++ ] = array[ leftIndex++ ];
89       else
90          tempArray[ combinedIndex++ ] = array[ rightIndex++ ];
91    } /* конец while */
92
93    if ( leftIndex == middle2 ) { /* достигнут конец левой половины ... */
94       while ( rightIndex <= right ) /* скопировать правую половину */
95          tempArray[ combinedIndex++ ] = array[ rightIndex++ ];
96    } /* конец if */
97    else {                        /* достигнут конец правой половины... */
98       while ( leftIndex <= middle1 ) /* скопировать левую половину */
99          tempArray[ combinedIndex++ ] = array[ leftIndex++ ];
100   } /* конец else */
101
102   /* скопировать значения обратно в оригинальный массив */
103   for ( i = left; i <= right; i++ )
104      array[ i ] = tempArray[ i ];
105
106   /* вывести объединенный массив */
107   printf( "%s* ","          ");
108   displaySubArray( array, left, right );
109   puts( "\n" );
110 } /* конец функции merge */
111
112 /* выводит элементы массива */
113 void displayElements( int array[], int length )
114 {
115    displaySubArray( array, 0, length - 1 );
116 } /* конец функции displayElements */
117
118 /* выводит определенные элементы массива */
```

```
119 void displaySubArray( int array[], int left, int right )
120 {
121    int i; /* для использования в цикле for */
122
123    /* вывести пробелы для выравнивания */
124    for ( i = 0; i < left; i++ )
125       printf( "%s* ", "   ");
126
127    /* вывести элементы левой половины */
128    for ( i = left; i <= right; i++ )
129       printf( " %d", array[ i ] );
130 } /* конец функции displaySubArray */
```

```
Unsorted array:
 79  86  60  79  76  71  44  88  58  23

split:       79  86  60  79  76  71  44  88  58  23
             79  86  60  79  76
                                 71  44  88  58  23
split:       79  86  60  79  76
             79  86  60
                         79  76
split:       79  86  60
             79  86
                 60
split:       79  86             79
                 86
merge:       79
                 86
             79  86
merge:       79  86
                 60
             60  79  86
split:               79  76
                     79
                         76
merge:               79
                         76
                     76 79
merge:       60  79  86
                     76  79
             60  76  79  79  86
split:                           71  44  88  58  23
                                 71  44  88
                                             58  23
split:                           71  44  88
                                 71  44
                                         88
split:                           71  44
                                 71
                                     44
merge:                           71
                                     44
                                 44  71
merge:                           44  71
                                        88
                                 44  71  88
split:                                      58  23
```

```
                                                  58
                                                      23
merge:                                            58
                                                      23
                                                  23  58
merge:                              44  71  88
                                                  23  58
                                    23  44  58  71  88
merge:        60  76  79  79  86
                                    23  44  58  71  88
              23  44  58  60  71  76  79  79  86  88

Sorted array:
 23  44  58  60  71  76  79  79  86  88
```

Цикл в строках 84–91 внутри функции `merge` выполняется, пока не будет достигнут конец одной из половинок массива. В строке 87 определяется, элемент какого массива меньше. Если меньше оказывается элемент в левом массиве, в строке 88 он помещается в следующую позицию в объединенном массиве. Если меньше оказывается элемент в правом массиве, тогда он помещается в следующую позицию в объединенном массиве, в строке 90. Когда цикл `while` завершит работу, одна из половин целиком окажется в объединенном массиве, но в другой половине останутся еще необработанные данные. В строке 93 проверяется, был ли достигнут конец в левой половине, и если условие выполняется, в строках 94–95 производится копирование остальных элементов из правой половины в объединенный массив. Если конец левой половины не был достигнут, следовательно, был достигнут конец правой половины, и в строках 98–99 производится копирование остальных элементов из левой половины. В заключение в строках 103–104 объединенный массив копируется в исходный. Вывод программы показывает, как происходят разбиение и слияние на каждом этапе сортировки.

Эффективность алгоритма сортировки методом слияния

Алгоритм сортировки методом слияния намного эффективнее алгоритмов сортировки методами выбора и вставки (хотя в это верится с трудом после взгляда на длинный листинг примера D.3). Рассмотрим первый (нерекурсивный) вызов функции `sortSubArray`. Он выполняет два рекурсивных вызова `sortSubArray` с двумя массивами, размер каждого из которых примерно равен половине исходного массива, и единственный вызов функции `merge`. Этот вызов функции `merge` выполнит, в худшем случае, $n - 1$ сравнений, что дает нам сложность $O(n)$. (Напомню, что каждый элемент для объединенного массива выбирается путем сравнения двух элементов в половинах исходного массива.) Два вызова функции `sortSubArray` произведут четыре рекурсивных вызова `sortSubArray` с массивами, размеры которых составляют примерно четверть исходного массива, а также два вызова функции `merge`. Эти два вызова `merge` выполнят, в худшем случае, $n/2 - 1$ сравнений

с общим числом сравнений $O(n)$. Далее каждый вызов `sortSubArray` еще два раза вызовет `sortSubArray` и один раз `merge`, и так будет продолжаться, пока исходный массив не окажется разбит на две половинки, содержащие по одному элементу. На каждом уровне будет выполнено $O(n)$ сравнений для объединения половин массива. На каждом уровне массив разбивается пополам, поэтому увеличение размера массива в два раза добавит лишь одно дополнительное разбиение. Увеличение размера массива в четыре раза добавит два дополнительных разбиения. В результате получается логарифмическая зависимость количества уровней разбиения от размера массива: $\log_2 n$. Что дает общую сложность алгоритма $\boldsymbol{O(n \log n)}$.

В табл. D.1 перечислены некоторые алгоритмы поиска и сортировки, описанные в этой книге, и их сложность в нотации «Большого О». В табл. D.2 перечислены значения в нотации «Большого О», упоминавшиеся в этом разделе с количественными характеристиками для n, чтобы нагляднее показать темпы роста количества операций, которые требуется выполнить для обработки n элементов.

Таблица D.1 | Алгоритмы поиска и сортировки и их значения «Большого О»

Алгоритм	«Большое О»
Сортировка вставкой	$O(n^2)$
Сортировка выбором	$O(n^2)$
Сортировка слиянием	$O(n \log n)$
Пузырьковая сортировка	$O(n^2)$
Быстрая сортировка	В худшем случае: $O(n^2)$ В среднем: $O(n \log n)$

Таблица D.2 | Примерное количество сравнений для часто встречающихся значений нотации «Большого О»

n	Примерное десятичное значение	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$
2^{10}	1000	10	2^{10}	10×2^{10}	2^{20}
2^{20}	1 000 000	20	2^{20}	20×2^{20}	2^{40}
2^{30}	1 000 000 000	30	2^{30}	30×2^{30}	2^{60}

E

Дополнительные особенности стандарта C

В этом приложении вы узнаете:

- о дополнительных ключевых особенностях стандартов C99 и C11;
- как объединять объявления и выполняемый код, как объявлять переменные в заголовке инструкции `for`;
- как инициализировать массивы и структуры с помощью инициализаторов;
- как пользоваться типом данных `bool` для создания логических переменных, которые могут иметь только два значения, `true` и `false`;
- как обслуживать массивы с переменной длиной;
- как выполнять арифметические операции с комплексными значениями.

Краткое содержание

E.1 Введение

C99 (1999) и **C11** (2011) являются ревизиями стандарта языка программирования C, совершенствующими существующие и добавляющими новые особенности. В ревизию C99 стандарта было внесено намного больше изменений, чем в C11. Знакомясь со сведениями в этом разделе, помните, что не все компиляторы поддерживают в полном объеме нововведения, появившиеся в ревизии C99 стандарта. Кроме того, ревизия C11 до сих пор остается достаточно новой, и многие компиляторы пока не поддерживают ее. Перед использованием особенностей, описываемых здесь, проверьте, поддерживаются ли они вашим компилятором. Цель данного приложения состоит лишь в том, чтобы познакомить вас с этими особенностями и подсказать ресурсы для дальнейшего чтения.

Мы обсудим здесь и дадим ссылки на некоторые свободно распространяемые компиляторы и IDE, обеспечивающие в той или иной мере поддержку ревизий C99 и C11 стандарта. Описания некоторых из представленных особенностей будут сопровождаться примерами действующего программного кода, включая возможность смешивания объявлений и выполняемого кода, поддержку объявлений переменных в инструкции `for`, назначенные ини-

циализаторы (designated initializers), составные литералы, тип `bool`, неявный тип `int` возвращаемого значения в прототипах и определениях функций (не допускается в C11), комплексные числа и массивы переменной длины. Мы также дадим краткое описание отдельных ключевых особенностей C99, включая ограниченные указатели (restricted pointers), надежное целочисленное деление (reliable integer division), гибкие члены-массивы, обобщенные математические операции, встраивание функций и инструкцию `return` без выражения. Другой важной особенностью ревизии C99 стандарта является добавление в `<math.h>` версий большинства математических функций, возвращающих значения типов `float` и `long double`. Мы обсудим возможности, определяемые последней ревизией C11 стандарта, включая поддержку многопоточной модели выполнения, улучшенную поддержку Unicode®, спецификатор `_Noreturn` функций, выражения обобщенного типа, функцию `quick_exit`, анонимные структуры и объединения, управление выравниванием в памяти, статические утверждения, анализируемость и вещественные типы. Многие из этих особенностей определены как необязательные. Мы включили также обширный список ресурсов в Интернете, где можно найти компиляторы и интегрированные среды разработки (IDE) с поддержкой C11, а еще дополнительные технические подробности о языке.

E.2 Поддержка положений ревизии C99

Большинство компиляторов C и C++ не поддерживали положения ревизии C99 стандарта на момент ее выпуска. Но с годами положение дел улучшилось, и многие современные компиляторы достаточно полно поддерживают C99. Однако Microsoft Visual C++ не поддерживает C99. Более подробную информацию по этой теме можно найти по адресу: `http://blogs.msdn.com/vcblog/archive/2007/11/05/iso-c-standard-update.aspx`.

В этом приложении мы использовали компилятор GNU GCC 4.3 в ОС Linux, поддерживающий большую часть особенностей, определяемых ревизией C99. Чтобы указать, что компиляция должна выполняться в соответствии с положениями стандарта C99, при компиляции программ необходимо добавлять аргумент командной строки `-std=c99`. Пользователи Windows также могут применять компилятор GCC, загрузив и установив либо пакет Cygwin (`www.cygwin.com`), либо MinGW (`sourceforge.net/projects/mingw`). Cygwin – это UNIX-подобная среда для Windows, а MinGW (Minimalist GNU for Windows) – версия компилятора и сопутствующих инструментов, созданная специально для Windows.

E.3 Заголовочные файлы в C99

В табл. E.1 перечислены стандартные заголовочные файлы в алфавитном порядке, которые добавлены в ревизии C99 (три из них были добавлены в ревизии C95). Все эти заголовочные файлы также остаются доступными

в ревизии C11. Новые заголовочные файлы, появившиеся в ревизии C11, будут перечислены ниже в этом приложении.

Таблица E.1 | Стандартные заголовочные файлы, добавленные в ревизиях C99 и C95

Стандартные заголовочные файлы	Описание
`<complex.h>`	Содержит макросы и прототипы функций поддержки комплексных чисел (см. раздел E.9). [Добавлен в ревизии C99]
`<fenv.h>`	Предоставляет информацию о реализации и возможностях поддержки вещественных чисел. [Добавлен в ревизии C99]
`<inttypes.h>`	Определяет несколько новых переносимых целочисленных типов и спецификаторов форматов для них. [Добавлен в ревизии C99]
`<iso646.h>`	Определяет макросы, позволяющие использовать поразрядные и логические операторы языка C, которые не могут быть быстро или легко напечатаны на некоторых интернациональных и не *QWERTY*-клавиатурах. [Добавлен в ревизии C95]
`<stdbool.h>`	Содержит макросы, определяющие тип `bool`, а также значения `true` и `false`, используемые для объявления логических переменных и выполнения операций с ними (см. раздел E.7). [Добавлен в ревизии C99]
`<stdint.h>`	Определяет дополнительные целочисленные типы и макросы для работы с ними. [Добавлен в ревизии C99]
`<tgmath.h>`	Определяет обобщенные макросы, позволяющие использовать функции из `<math.h>` с параметрами разных типов (см. раздел E.12). [Добавлен в ревизии C99]
`<wchar.h>`	В комплексе с заголовочным файлом `<wctype.h>` обеспечивает поддержку ввода/вывода многобайтных символов. [Добавлен в ревизии C95]
`<wctype.h>`	В комплексе с заголовочным файлом `<wcchar.h>` обеспечивает поддержку ввода/вывода многобайтных символов. [Добавлен в ревизии C95]

E.4 Включение объявлений в выполняемый код

[**Этот раздел следует читать после раздела 2.3.**]
До выхода ревизии C99 *все* переменные должны были объявляться в начале блока их видимости. Ревизия C99 позволяет смешивать объявления и выполняемый код. Переменная может быть объявлена в любом месте блока до ее использования. Взгляните на программу в примере E.1.

Пример E.1 | Смешивание объявлений переменных и выполняемого кода

```
 1 // Пример E.1: figE_02.c
 2 // Смешивание объявлений переменных и выполняемого кода
 3 #include <stdio.h>
 4
 5 int main( void )
 6 {
 7    int x = 1; // объявление переменной в начале блока
 8    printf( "x is %d\n", x );
 9
10    int y = 2; // объявление переменной среди выполняемого кода
11    printf( "y is %d\n", y );
12 } // конец main
```

```
x is 1
y is 2
```

В этой программе сначала вызывается функция `printf` (выполняемый код) в строке 8, а затем в строке 10 объявляется переменная `y` типа `int`. Согласно положениям ревизии C99, допускается объявлять переменные непосредственно перед их использованием, даже если эти объявления окажутся *среди* выполняемого кода. Нет нужды объявлять переменную `int y` (строка 10) заранее, до того, как она потребуется (в строке 11). Эта возможность может улучшить читаемость программы и уменьшить вероятность обращений к ошибочным переменным, однако некоторые программисты предпочитают объявлять переменные в начале блоков. Имейте в виду, что переменные не могут использоваться до их объявления.

E.5 Объявление переменных в заголовках инструкций for

[**Этот раздел следует читать после раздела 4.4.**]
Как вы, наверное, помните, инструкция `for` состоит из выражения инициализации, условия продолжения цикла, выражения инкремента и тела цикла.

Ревизия C99 позволяет включать объявление переменной в выражение инициализации инструкции `for`. Для использования в качестве счетчика цикла можно создать новую переменную, объявив ее в заголовке инструкции `for`, при этом область видимости такой переменной будет ограничена инструкцией `for`. Программа в примере E.2 объявляет переменную в заголовке инструкции `for`.

Пример E.2 | Объявление переменной в заголовке инструкции for

```
 1 // Пример E.2: figE_03.c
 2 // Объявление переменной в заголовке инструкции for
 3 #include <stdio.h>
 4
```

```
 5 int main( void )
 6 {
 7    printf( "Values of x\n" );
 8
 9    // объявление переменной в заголовке инструкции for
10    for ( int x = 1; x <= 5; ++x ) {
11       printf( "%d\n", x );
12    } // конец for
13 } // конец main
```

```
Values of x
1
2
3
4
5
```

Любая переменная, объявленная в заголовке инструкции `for`, будет доступна только в этой инструкции – к такой переменной нельзя будет обратиться *из-за пределов инструкции* `for`, так как подобная попытка вызовет ошибку компиляции.

E.6 Назначенные инициализаторы и составные литералы

[Этот раздел следует читать после раздела 10.3.]
Назначенные инициализаторы дают возможность инициализировать элементы массивов, объединений и структур по их индексам или именам. В примере E.3 показано, как можно присвоить значения первому и последнему элементам массива.

Пример E.3 | Инициализация элементов массива при использовании компилятора, не поддерживающего C99

```
 1 /* Пример E.3: figE_04.c
 2 Инициализация элементов массива при использовании компилятора,
 3 не поддерживающего C99 */
 4 #include <stdio.h>
 5
 6 int main( void )
 7 {
 8    int i;      /* объявление счетчика цикла */
 9    int a[ 5 ]; /* объявление массива */
10
11    a[ 0 ] = 1; /* явное присваивание значений элементам массива... */
12    a[ 4 ] = 2; /* после его объявления */
13
14    /* присвоить ноль всем элементам массива, кроме первого и последнего */
15    for ( i = 1; i < 4; ++i ) {
16       a[ i ] = 0;
17    } /* конец for */
```

```
18
19    /* вывести содержимое массива */
20    printf( "The array is\n" );
21
22    for ( i = 0; i < 5; ++i ) {
23       printf( "%d\n", a[ i ] );
24    } /* конец for */
25 } /* конец main */
```

```
The array is
1
0
0
0
2
```

В примере E.4 приводится аналогичная программа, где также выполняется явная инициализация первого и последнего элементов массива, но на этот раз с помощью назначенного инициализатора.

Пример E.4 | Использование назначенного инициализатора для инициализации элементов массива в C99

```
 1 // Пример E.4: figE_05.c
 2 // Использование назначенного инициализатора
 3 // для инициализации элементов массива в C99
 4 #include <stdio.h>
 5
 6 int main( void )
 7 {
 8    int a[ 5 ] =
 9    {
10       [ 0 ] = 1, // инициализировать элементы с помощью назначенного
11       [ 4 ] = 2  // инициализатора внутри объявления массива
12    }; // точка с запятой обязательна
13
14    // вывести содержимое массива
15    printf( "The array is \n" );
16
17    for ( int i = 0; i < 5; ++i ) {
18       printf( "%d\n", a[ i ] );
19    } // конец for
20 } // конец main
```

```
The array is
1
0
0
0
2
```

В строках 8–12 объявляется массив, и в фигурных скобках инициализируются указанные элементы этого массива. Обратите внимание на синтаксис. Каждый инициализатор в списке (строки 10–11) отделяется от следующего

запятой, а за закрывающей фигурной скобкой следует точка с запятой. Элементы, которые не инициализируются явно, получают нулевые значения (требуемого типа). Этот синтаксис был недопустим до выхода ревизии C99.

Кроме объявлений переменных, списки инициализаторов можно также использовать для инициализации анонимных массивов, структур и объединений. В этом случае их часто называют **составными литерами**. Например, если потребуется передать функции массив, такой как в примере E.4, без объявления дополнительной переменной, можно использовать следующий прием:

```
demoFunction( ( int [ 5 ] ) { [ 0 ] = 1, [ 4 ] = 2 } );
```

Взгляните на более сложный пример E.5, где назначенные инициализаторы используются для инициализации массива структур.

Пример E.5 | Использование назначенного инициализатора для инициализации массива структур в C99

```
// Пример E.5: figE_06.c
// Использование назначенного инициализатора
// для инициализации массива структур в C99
#include <stdio.h>

struct twoInt // объявление структуры с двумя целыми числами
{
   int x;
   int y;
}; // конец структуры struct twoInt

int main( void )
{
   // явная инициализация элементов массива a
   // с явной инициализацией обоих членов структур в элементах
   struct twoInt a[ 5 ] =
   {
      [ 0 ] = { .x = 1, .y = 2 },
      [ 4 ] = { .x = 10, .y = 20 }
   };

   // вывести содержимое массива
   printf( "x\ty\n" );

   for ( int i = 0; i < 5; ++i ) {
      printf( "%d\t%d\n", a[ i ].x, a[ i ].y );
   } // конец for
} // конец main
```

```
x y
1 2
0 0
0 0
0 0
10 20
```

В строке 18 используется *назначенный инициализатор*, явно инициализирующий элементы массива структурами. Внутри него применяется вложенный назначенный инициализатор, явно инициализирующий члены `x` и `y` структуры. Для инициализации членов структуры или объединения имена членов должны начинаться с символа точки.

Сравните строки 16–20 в примере E.5, где применяется назначенный инициализатор, со следующим выполняемым кодом, не использующим назначенных инициализаторов:

```
struct twoInt a[ 5 ];
a[ 0 ].x = 1;
a[ 0 ].y = 2;
a[ 4 ].x = 10;
a[ 4 ].y = 20;
```

E.7 Тип bool

[**Этот раздел следует читать после раздела 3.4.**]
Ревизия C99 определяет **логический тип** `_Bool`. Переменные этого типа могут хранить только значения 0 и 1. В соответствии с соглашениями, принятыми в языке C, *нулевые* и *ненулевые* значения представляют *ложь* и *истину* – значение 0 в условиях интерпретируется как *ложь*, а *любое* ненулевое значение – как *истина*. При попытке присвоить *любое* ненулевое значение переменной типа `_Bool` она будет установлена в значение 1. Ревизия C99 определяет заголовочный файл `<stdbool.h>`, определяющий макросы, представляющие тип `bool` и его значения (`true` и `false`). Эти макросы замещают `true` числом 1, `false` – числом 0 и `bool` – ключевым словом `_Bool`. В примере E.6 используется функция `isEven` (строки 29–37), возвращающая логическое значение `true`, если в аргументе ей было передано четное число, и `false` – если нечетное.

Пример E.6 | Использование типа bool и значений true и false

```
 1 // Пример E.6: figE_07.c
 2 // Использование типа bool и значений true и false в стандарте C99.
 3 #include <stdio.h>
 4 #include <stdbool.h> // включить поддержку bool, true и false
 5
 6 bool isEven( int number ); // прототип функции
 7
 8 int main( void )
 9 {
10    // цикл по двум введенным значениям
11    for ( int i = 0; i < 2; ++i ) {
12       int input; // значение, введенное пользователем
13       printf( "Enter an integer: " );
14       scanf( "%d", &input );
15
16       bool valueIsEven = isEven( input ); // проверить четность
17
```

```
18       // сообщить о четности или нечетности введенного числа
19       if ( valueIsEven ) {
20          printf( "%d is even \n\n", input );
21       } // конец if
22       else {
23          printf( "%d is odd \n\n", input );
24       } // конец else
25    } // конец for
26 } // конец main
27
28 // возвращает true для четных чисел
29 bool isEven( int number )
30 {
31    if ( number % 2 == 0 ) { // число делится на 2 без остатка?
32       return true;
33    }
34    else {
35       return false;
36    }
37 } // конец функции isEven
```

```
Enter an integer: 34
34 is even
Enter an integer: 23
23 is odd
```

В строке 16 объявляется переменная типа `bool` с именем `valueIsEven`. В строках 13–14, в цикле, запрашивается и вводится следующее целое число. В строке 16 введенное число передается функции `isEven` (определена в строках 29–37). Функция `isEven` возвращает значение типа `bool`. В строке 31 проверяется, делится ли аргумент на 2 без остатка. Если делится (то есть если число *четное*), инструкция `return` в строке 32 возвращает значение `true`; в противном случае (то есть если число *нечетное*) инструкция `return` в строке 35 возвращает `false`. Результат присваивается логической переменной `valueIsEven` в строке 16. Если переменная `valueIsEven` получает значение `true`, в строке 20 выводится сообщение о том, что введенное число – *четное*. Если `valueIsEven` получает значение `false`, в строке 23 выводится сообщение о том, что введенное число – *нечетное*.

E.8 Неявный тип int в объявлениях функций

[Этот раздел следует читать после раздела 5.5.]
В ревизиях стандарта языка C, предшествовавших C99, если в объявлении функции не определялся *явно* тип возвращаемого значения, *неявно* предполагалось, что она возвращает значение типа `int`. Кроме того, если тип параметра не определялся явно, предполагалось, что он имеет тип `int`. Взгляните на программу в примере E.7.

Пример E.7 | Использование неявного типа int в ревизиях, предшествовавших C99

```
 1 /* Пример E.7: figE_08.c
 2 Использование неявного типа int в ревизиях, предшествовавших C99 */
 3 #include <stdio.h>
 4
 5 returnImplicitInt(); /* прототип без типа возвращаемого значения */
 6 int demoImplicitInt( x ); /* прототип без типа параметра */
 7
 8 int main( void )
 9 {
10    int x;
11    int y;
12
13    /* присвоить значение, возвращаемое функцией с необъявленным типом */
14    x = returnImplicitInt();
15
16    /* передать целое число функции с параметром без типа */
17    y = demoImplicitInt( 82 );
18
19    printf( "x is %d\n", x );
20    printf( "y is %d\n", y );
21 } /* конец main */
22
23 returnImplicitInt()
24 {
25    return 77; /* возвращает int, если тип не объявлен явно */
26 } /* конец функции returnImplicitInt */
27
28 int demoImplicitInt( x )
29 {
30    return x;
31 } /* конец функции demoImplicitInt */
```

Если попытаться скомпилировать эту программу с помощью компилятора Microsoft Visual C++, который несовместим с положениями ревизии C99, никаких ошибок во время компиляции обнаружено не будет, и программа будет работать правильно. Ревизия C99 *запрещает* неявное использование типа `int` и требует, чтобы компиляторы реагировали на подобные объявления выводом сообщения об ошибке или предупреждения. Если попытаться скомпилировать эту программу с помощью компилятора GCC 4.7, он выведет предупреждения, как показано в примере E.8.

Пример E.8 | Предупреждения компилятора GCC 4.3 об использовании неявного объявления типа int

```
figE_08.c:5:1: warning: data definition has no type or storage class
   [enabled by default]
figE_08.c:5:1: warning: type defaults to 'int' in declaration of
   'returnImplicitInt' [enabled by default]
figE_08.c:6:1: warning: parameter names (without types) in function
   declaration [enabled by default]
figE_08.c:23:1: warning: return type defaults to 'int' [enabled by
default]
figE_08.c: In function 'demoImplicitInt':
figE_08.c:28:5: warning: type of 'x' defaults to 'int' [enabled by
default]
```

E.9 Комплексные числа

[**Этот раздел следует читать после раздела 5.3.**]
Ревизия C99 стандарта определяет поддержку комплексных чисел и арифметических операций с ними. Программа в примере E.9 выполняет простейшие операции с комплексными числами.

Пример E.9 | Поддержка комплексных чисел в C99

```
 1 // Пример E.9: figE_10.c
 2 // Поддержка комплексных чисел в C99
 3 #include <stdio.h>
 4 #include <complex.h> // определение типа complex и математических функций
 5
 6 int main( void )
 7 {
 8    double complex a = 32.123 + 24.456 * I; // значение 32.123 + 24.456i
 9    double complex b = 23.789 + 42.987 * I; // значение 23.789 + 42.987i
10    double complex c = 3.0 + 2.0 * I;
11
12    double complex sum = a + b;        // сложение комплексных чисел
13    double complex pwr = cpow( a, c ); // возведение в степень
14
15    printf( "a is %f + %fi\n", creal( a ), cimag( a ));
16    printf( "b is %f + %fi\n", creal( b ), cimag( b ));
17    printf( "a + b is: %f + %fi\n", creal( sum ), cimag( sum ));
18    printf( "a - b is: %f + %fi\n", creal( a - b ), cimag( a - b ));
19    printf( "a * b is: %f + %fi\n", creal( a * b ), cimag( a * b ));
20    printf( "a / b is: %f + %fi\n", creal( a / b ), cimag( a / b ));
21    printf( "a ^ b is: %f + %fi\n", creal( pwr ), cimag( pwr ));
22 } // конец main
```

```
a is 32.123000 + 24.456000i
b is 23.789000 + 42.987000i
a + b is: 55.912000 + 67.443000i
a - b is: 8.334000 + -18.531000i
a * b is: -287.116025 + 1962.655185i
a / b is: 0.752119 + -0.331050i
a ^ b is: -17857.051995 + 1365.613958i
```

Чтобы добавить поддержку комплексных чисел, предусматриваемую ревизией C99, мы подключили заголовочный файл `<complex.h>` (строка 4). В нем определяется макрос `complex`, разворачивающийся в ключевое слово `_Complex` – тип, резервирующий массив с двумя элементами, соответствующими *действительной и мнимой частям* комплексного числа.

Подключив заголовочный файл в строке 4, мы получили возможность определять переменные, как показано в строках 8–10 и 12–13. Каждая переменная – `a`, `b`, `c`, `sum` и `pwr` – определяется с типом `double complex`. Также можно было бы использовать типы `float complex` и `long double complex`.

Арифметические операторы могут применяться и к комплексным числам. Заголовочный файл `<complex.h>` дополнительно содержит прототипы нескольких математических функций, например `cpow`, которая вызывается

в строке 13. Допускается также использовать операторы !, ++, --, &&, ||, ==, != и унарный &.

В строках 17–21 выводятся результаты различных арифметических операций. Действительную и мнимую части комплексного числа можно извлечь с помощью функций `creal` и `cimag` соответственно, как показано в строках 15–21. В сообщении, что выводится в строке 21, мы использовали символ ^, чтобы показать возведение в степень.

E.10 Массивы переменной длины

[Этот раздел следует читать после раздела 6.9.]

До выхода ревизии C99 массивы могли иметь только *постоянный размер*. Но как быть, если размер массива станет известен лишь во время выполнения? В таких ситуациях вы вынуждены были бы задействовать *механизмы управления динамической памятью* в виде функции `malloc` и др. Ревизия C99 позволяет определять массивы заранее неизвестного размера, используя *массивы переменной длины* (Variable-Length Arrays, VLA). Массив переменной длины – это массив, длина или размер которого определяется как выражение, вычисляемое во время выполнения. Программа в примере E.10 объявляет и выводит несколько массивов VLA.

Пример E.10 | Использование массивов переменной длины

```
 1 // Пример E.10: figE_11.c
 2 // Использование массивов переменной длины
 3 #include <stdio.h>
 4
 5 // прототипы функций
 6 void print1DArray( int size, int arr[ size ] );
 7 void print2DArray( int row, int col, int arr[ row ][ col ] );
 8
 9 int main( void )
10 {
11    int arraySize; // размер 1-мерного массива
12    int row1, col1, row2, col2; // количество строк и столбцов
13                                // в 2-мерных массивах
14    printf( "Enter size of a one-dimensional array: " );
15    scanf( "%d", &arraySize );
16
17    printf( "Enter number of rows and columns in a 2-D array: " );
18    scanf( "%d %d", &row1, &col1 );
19
20    printf( "Enter number of rows and columns in another 2-D array: " );
21    scanf( "%d %d", &row2, &col2 );
22
23    int array[ arraySize ];       // объявить 1-мерный массив VLA
24    int array2D1[ row1 ][ col1 ]; // объявить 2-мерный массив VLA
25    int array2D2[ row2 ][ col2 ]; // объявить 2-мерный массив VLA
```

```
26
27    // проверить действие оператора sizeof с массивами VLA
28    printf( "\nsizeof(array) yields array size of %d bytes\n",
29       sizeof( array ) );
30
31    // присвоить значения элементам 1-мерного массива
32    for ( int i = 0; i < arraySize; ++i ) {
33       array[ i ] = i * i;
34    } // конец for
35
36    // присвоить значения элементам 2-мерного массива
37    for ( int i = 0; i < row1; ++i ) {
38       for ( int j = 0; j < col1; ++j ) {
39          array2D1[ i ][ j ] = i + j;
40       } // конец for
41    } // конец for
42
43    // присвоить значения элементам второго 2-мерного массива
44    for ( int i = 0; i < row2; ++i ) {
45       for ( int j = 0; j < col2; ++j ) {
46          array2D2[ i ][ j ] = i + j;
47       } // конец for
48    } // конец for
49
50    printf( "\nOne-dimensional array:\n" );
51    print1DArray( arraySize, array ); // передать 1-мерный массив
52
53    printf( "\nFirst two-dimensional array:\n" );
54    print2DArray( row1, col1, array2D1 ); // передать 2-мерный массив
55
56    printf( "\nSecond two-dimensional array:\n" );
57    print2DArray( row2, col2, array2D2 ); // передать второй 2-мерный массив
58 } // конец main
59
60 void print1DArray( int size, int array[ size ] )
61 {
62    // вывести содержимое массива
63    for ( int i = 0; i < size; ++i ) {
64       printf( "array[%d] = %d\n", i, array[ i ] );
65    } // конец for
66 } // конец функции print1DArray
67
68 void print2DArray( int row, int col, int arr[ row ][ col ] )
69 {
70    // вывести содержимое массива
71    for ( int i = 0; i < row; ++i ) {
72       for ( int j = 0; j < col; ++j ) {
73          printf( "%5d", arr[ i ][ j ] );
74       } // конец for
75
76       printf( "\n" );
77    } // конец for
78 } // конец функции print2DArray
```

```
Enter size of a one-dimensional array: 6
Enter number of rows and columns in a 2-D array: 2 5
Enter number of rows and columns in another 2-D array: 4 3

sizeof(array) yields array size of 24 bytes

One-dimensional array:
array[0] = 0
array[1] = 1
array[2] = 4
array[3] = 9
array[4] = 16
array[5] = 25

First two-dimensional array:
    0    1    2    3    4
    1    2    3    4    5

Second two-dimensional array:
    0    1    2
    1    2    3
    2    3    4
    3    4    5
```

Сначала программа запрашивает у пользователя желаемые размеры одномерного и двух двумерных массивов (строки 14–21). Затем в строках 23–25 объявляются массивы переменной длины соответствующих размеров. Прежде такой прием приводил к ошибкам компиляции, но в компиляторах с поддержкой C99 он больше не вызывает ошибок, если переменные, представляющие размеры массивов, имеют целочисленный тип.

После объявления массивов мы применяем оператор `sizeof` (строка 29), чтобы убедиться, что наш массив переменной длины имеет требуемый размер. Прежде оператор `sizeof` выполнялся только на этапе компиляции, но при применении к массивам переменной длины он определяет размер аргумента на этапе выполнения. В выводе программы видно, что оператор `sizeof` вернул размер 24 байта – число, в четыре раза большее, чем указанное количество элементов, потому что размер одного элемента (типа `int`) на компьютере, где проводились испытания, составляет 4 байта.

Далее (строки 32–48) выполняется присваивание значений элементам массивов переменной длины. В качестве условия продолжения цикла, при заполнении одномерного массива, использовано выражение `i < arraySize`. Подобно фиксированным массивам, массивы переменной длины *не предоставляют никаких средств, препятствующих выходу за пределы массива.*

В строках 60–66 определяется функция `print1DArray`, принимающая одномерный массив переменной длины. Для передачи массива переменной длины функциям используется точно такой же синтаксис, как и для передачи фиксированных массивов. В объявлении параметра `array` мы использовали переменную `size` (`int array[size]`), но компилятор не выпол-

няет никаких дополнительных проверок, кроме проверки переменной на принадлежность к целочисленному типу – это всего лишь *напоминание* для программиста.

Функция `print2DArray` (строки 68–78) принимает двумерный массив переменной длины и выводит его содержимое. Вспомните, как в разделе 6.9 говорилось, что до выхода ревизии C99 необходимо было явно указывать все размерности массива, кроме первой. То же требование действует и для многомерных массивов переменной длины в C99, с той лишь разницей, что размеры могут определяться переменными. Начальное значение `col`, передаваемое функции, используется для преобразования двумерных индексов в смещения в непрерывной области памяти, как и в случае с фиксированными массивами. Изменение значения `col` за пределами функции не вызовет никаких изменений в индексировании элементов, но если передать измененное значение в функцию, это приведет к ошибке.

E.11 Дополнительные возможности препроцессора

[Этот раздел следует читать после главы 13.]

Ревизия C99 добавляет новые особенности в препроцессор C. Первой из них является оператор `_Pragma`, который действует подобно директиве `#pragma`, представленной в разделе 13.6. Вызов оператора `_Pragma("tokens")` оказывает такое же действие, как и директива `#pragma tokens`, но, в отличие от директивы, этот оператор можно использовать в определениях макросов. То есть, вместо того чтобы окружать директивами `#if` все прагмы, характерные для разных компиляторов, можно один раз определить макрос с оператором `_Pragma` и использовать его по всей программе.

Вторая особенность – три новые прагмы, определяющие поведение операций с вещественными числами. Первой лексемой в этих стандартных прагмах всегда является `STDC`, второй – одна из `FENV_ACCESS`, `FP_CONTRACT` или `CX_LIMITED_RANGE`, и третьей – `ON`, `OFF` или `DEFAULT`, которая указывает, должна ли данная прагма *включить* данную характеристику, *выключить* ее или *установить в значение по умолчанию* соответственно. Прагма `FENV_ACCESS` сообщает компилятору, какая часть кода будет использовать функции из заголовочного файла `<fenv.h>`. В современных настольных системах вещественная арифметика выполняется с 80-разрядными вещественными числами. Если включить характеристику `FP_CONTRACT`, компилятор будет выполнять последовательности операций с данной точностью и сохранять конечные результаты с более низкой точностью, в виде значения типа `float` или `double`, вместо того чтобы снижать точность после каждой операции. Наконец, если включить характеристику `CX_LIMITED_RANGE`, компилятору будет позволено использовать стандартные математические

формулы в операциях с комплексными числами, таких как умножение или деление. Поскольку вещественные числа не способны хранить точное представление некоторых чисел, использование обычных математических определений может приводить к переполнению, даже если операнды и конечный результат не превышают пределов представления вещественных чисел.

Третья новая особенность препроцессора позволяет передавать макросам *пустые аргументы* – в предыдущей ревизии стандарта языка C реакция макросов на пустые аргументы *не была определена*, хотя компилятор GCC действует в соответствии с положениями ревизии C99 даже в режиме C89. Во многих случаях попытка передать пустой аргумент вызывает синтаксическую ошибку, но иногда такой прием мог бы иметь практическую ценность. Например, представьте макрос `PTR(type, cv, name)`, объявленный как `type * cv name`. Иногда бывает нежелательно добавлять определение `const` или `volatile` в объявление указателя, поэтому второй аргумент может оказаться пустым. Когда пустой аргумент используется с оператором `#` или `##` (см. раздел 13.7), в результате получается пустая строка или идентификатор, являющийся результатом объединения аргументов соответственно.

Важнейшим расширением препроцессора стала поддержка списков аргументов переменной длины в макросах. Она позволяет создавать макросы-обертки вокруг функций, таких как `printf`, например с целью автоматического добавления имени текущего файла в инструкции отладки, как показано ниже:

```
#define DEBUG( ... ) printf( __FILE__ ": " __VA_ARGS__ )
```

Макрос `DEBUG` принимает переменное количество аргументов, как указывает троеточие `...` в списке аргументов. Как и в функциях, троеточие `...` должно завершать список аргументов; в отличие от функций, троеточие может быть единственным аргументом. Идентификатор `__VA_ARGS__`, начинающийся и завершающийся двумя символами подчеркивания, замещается фактическим списком аргументов. Если этот макрос вызвать, как показано ниже:

```
DEBUG( "x = %d, y = %d\n", x, y );
```

он будет развернут в вызов функции:

```
printf( "file.c" ": " "x = %d, y = %d\n", x, y );
```

Как упоминалось в разделе 13.7, литералы строк, разделенные пробельными символами, объединяются препроцессором в одну строку, поэтому три строковых литерала в примере выше будут объединены и переданы функции `printf` в первом аргументе.

E.12 Другие особенности, определяемые ревизией C99

В этом разделе представлен краткий обзор некоторых дополнительных особенностей, определяемых ревизией C99.

E.12.1 Минимальные требования компилятора к ресурсам

[**Этот раздел следует читать после раздела 14.5.**]
До выхода ревизии C99 стандарт языка C требовал от реализаций поддержку идентификаторов с длиной не менее 31 символа, которые используются для *внутреннего связывания* (внутри компилируемого файла), и не менее шести символов – в идентификаторах, используемых для *внешнего связывания* (на которые можно ссылаться из других файлов). Дополнительную информацию о внутреннем и внешнем связываниях можно получить в разделе 14.5. Ревизия C99 стандарта увеличила эти ограничения до 63 символов для внутреннего связывания и 31 символа – для внешнего. Однако это всего лишь *минимальные* требования. Компиляторы могут поддерживать *более длинные* идентификаторы. В настоящее время идентификаторы могут содержать национальные символы посредством механизма Universal Character Names (стандарт C99, раздел 6.4.3) и, по выбору создателей реализации, непосредственно (стандарт C99, раздел 6.4.2.1). [За дополнительной информацией обращайтесь к разделу 5.2.4.1 стандарта C99.]

В дополнение к увеличению длин идентификаторов, которые теперь обязаны поддерживать компиляторы, ревизия C99 стандарта устанавливает минимальные ограничения на многие другие особенности языка. Например, компиляторы обязаны поддерживать не менее 1023 членов в структурах, перечислениях и объединениях и не менее 127 параметров в функциях. За дополнительной информацией об этих и других ограничениях обращайтесь к разделу 5.2.4.1 стандарта C99.

E.12.2 Ключевое слово restrict

[**Этот раздел следует читать после раздела 7.5.**]
Ключевое слово `restrict` используется для объявления *ограниченных указателей*, когда указатель должен иметь *исключительный* доступ к области памяти. Объекты, доступ к которым осуществляется посредством ограниченных указателей, не должны быть доступны посредством других указателей, если только значения этих указателей не были получены из ограниченных указателей. Ниже показано, как объявить ограниченный указатель на переменную типа `int`:

```
int *restrict ptr;
```

Ограниченные указатели дают компилятору возможность оптимизировать доступ к памяти. Например, стандартная функция `memcpy` определена в ревизии C99 стандарта, как показано ниже:

```
void *memcpy( void *restrict s1, const void *restrict s2, size_t n);
```

В описании функции `memcpy` отмечается, что она не может использоваться для копирования *перекрывающихся* областей памяти. Использование ограниченных указателей позволяет компилятору заметить это требование и *оптимизировать* операцию копирования за счет копирования сразу нескольких байтов за раз. Некорректное объявление указателя ограниченным, когда имеется другой указатель, ссылающийся на ту же область памяти, может привести к *непредсказуемому поведению*. [За дополнительной информацией обращайтесь к разделу 6.7.3.1 стандарта C99.]

E.12.3 Надежное целочисленное деление

[**Этот раздел следует читать после раздела 2.4.**]
В компиляторах, не поддерживающих положения ревизии C99, поведение операции целочисленного деления различалось для разных реализаций. Некоторые реализации округляли отрицательное частное в сторону отрицательной бесконечности, другие – в сторону нуля. Если один из операндов оказывался отрицательным, это могло приводить к разным результатам. Рассмотрим деление –28 на 5. Точный результат равен –5.6. Если реализация выполняет округление в сторону отрицательной бесконечности, мы получим –6, если в сторону нуля – результат будет равен –5. Ревизия C99 устранила эту неоднозначность и *явно* требует, чтобы операция целочисленного деления (или определения остатка – деления по модулю) *выполняла округление в сторону нуля*. Это обеспечивает надежность целочисленного деления – все платформы, совместимые с положениями ревизии C99, будут возвращать одинаковые результаты. [За дополнительной информацией обращайтесь к разделу 6.5.5 стандарта C99.]

E.12.4 Гибкие члены-массивы

[**Этот раздел следует читать после раздела 10.3.**]
Ревизия C99 позволяет объявлять в последних членах структур массивы неопределенной длины. Взгляните на следующий пример:

```
struct s {
   int arraySize;
   int array[];
}; // конец структуры s
```

Гибкие члены-массивы объявляются с применением пустых квадратных скобок (`[]`). Для выделения памяти под структуру с гибким членом-массивом можно использовать следующий код:

```
int desiredSize = 5;
struct s *ptr;
ptr = malloc( sizeof( struct s ) + sizeof( int ) * desiredSize );
```

Оператор `sizeof` игнорирует гибкие члены-массивы. Выражение `sizeof ( struct s )` вернет суммарный размер всех членов структуры `struct s`, *кроме* последнего члена-массива. Дополнительное пространство определяется выражением `sizeof( int ) * desiredSize`, возвращающим размер гибкого члена-массива.

На применение гибких членов-массивов существует множество ограничений. Гибкий член-массив может быть *только последним членом структуры*, то есть любая структура может содержать *не более одного* гибкого члена-массива. Кроме того, гибкий член-массив не может быть единственным членом структуры. Структура должна иметь *хотя бы один* фиксированный член. Далее, любая структура, содержащая гибкий член-массив, *не может быть* членом другой структуры. Наконец, структура с гибким членом-массивом не может инициализироваться *статически* – такая структура должна распределяться *динамически*. Размер гибкого члена-массива нельзя зафиксировать на этапе компиляции. [За дополнительной информацией обращайтесь к разделу 6.7.2.1 стандарта C99.]

E.12.5 Ослабление ограничений в составных инициализаторах

[Этот раздел следует читать после раздела 10.3.]
В соответствии с ревизией C99 больше не требуется, чтобы переменные составных типов, таких как массивы, структуры и объединения, инициализировались константными выражениями. Это дает возможность использовать более краткие списки инициализаторов вместо множества отдельных инструкций, инициализирующих отдельные элементы.

E.12.6 Математические операции обобщенного типа

[Этот раздел следует читать после раздела 5.3.]
В ревизии C99 был добавлен новый заголовочный файл `<tgmath.h>`. Он содержит определения макросов вызова многих математических функций из `<math.h>` для выполнения операций с аргументами разных типов. Например, после подключения `<tgmath.h>` для переменной `x` типа `float` выражение `sin(x)` будет развернуто в вызов функции `sinf` (принимающей аргумент типа `float`); для переменной `x` типа `double` выражение `sin(x)` будет развернуто в вызов функции `sin` (принимающей аргумент типа `double`); для переменной `x` типа `long double` выражение `sin(x)` будет развернуто в вызов функции `sinl` (принимающей аргумент типа `long double`); и для переменной `x` типа `complex number` выражение `sin(x)` будет развернуто в вызов соответствующей версии функции `sin` комплексного типа (`csin`, `csinf` или `csinl`). Ревизия C11 включает дополнительные возможности обобщения типов, о которых будет рассказываться далее в этом разделе.

E.12.7 Встраиваемые функции

[Этот раздел следует читать после раздела 5.5.]
Ревизия C99 позволяет объявлять *встраиваемые функции* (inline functions) (подобно языку C++) с помощью ключевого слова `inline`, как показано ниже:

```
inline void randomFunction();
```

Это ключевое слово *не оказывает влияния* на логику программы с точки зрения пользователя, но может *повышать производительность программ*. На вызовы функций расходуется некоторое время. Когда функция объявляется как встраиваемая, ее тело подставляется компилятором в точку вызова, и программе не приходится тратить время на вызов такой функции. Этот прием увеличивает производительность во время выполнения, но может также увеличивать размер программы. Объявляйте функции встраиваемыми, *только* если они достаточно коротки и вызываются очень часто. Объявление `inline` носит *рекомендательный* характер для компилятора и может игнорироваться им. [За дополнительной информацией обращайтесь к разделу 6.7.4 стандарта C99.]

E.12.8 Инструкция return без выражения

[Этот раздел следует читать после раздела 5.5.]
Ревизия C99 определяет более строгие ограничения для инструкции возврата из функции. В функциях, возвращающих значение типа, отличного от `void`, не допускается более использовать инструкцию

```
return;
```

В компиляторах, несовместимых с ревизией C99, это допустимо, но может приводить к *непредсказуемым результатам*, если вызывающая программа попытается использовать возвращаемое значение. Аналогично в функциях, не возвращающих значения, нельзя больше возвращать какое-либо значение. Инструкции, такие как

```
void returnInt() { return 1; }
```

больше недопустимы. Ревизия C99 требует, чтобы совместимые с ней компиляторы генерировали предупреждения или сообщения об ошибках в подобных ситуациях. [За дополнительной информацией обращайтесь к разделу 6.8.6.4 стандарта C99.]

E.12.9 Предопределенный идентификатор __func__

[Этот раздел следует читать после раздела 13.5.]
Предопределенный идентификатор `__func__` действует подобно макросам `__FILE__` и `__LINE__` препроцессора – его значением является строка с *именем текущей функции*. В отличие от `__FILE__`, он является не строковым лите-

ралом, а настоящей переменной и потому не может объединяться с другими строковыми литералами. Это обусловлено тем, что конкатенация строковых литералов выполняется на этапе работы препроцессора, который ничего не знает о семантике языка C.

E.12.10 Макрос va_copy

[Этот раздел следует читать после раздела 14.3.]
В разделе 14.3 были представлены заголовочный файл `<stdarg.h>` и средства для работы со списками аргументов переменной длины. Ревизия C99 добавляет макрос `va_copy`, который принимает два списка `va_list` и копирует второй аргумент в первый. Это дает возможность несколько раз выполнить обход списка аргументов переменной длины, не начиная каждый раз все с самого начала.

E.13 Новые особенности в ревизии C11

При подготовке этой книги к публикации была утверждена ревизия C11 стандарта, расширяющая возможности языка C. На момент написания этих строк большинство компиляторов, заявленных как совместимые с положениями ревизии C11, обеспечивало поддержку лишь некоторых ее особенностей. Кроме того, множество особенностей в ревизии C11 рассматриваются как необязательные. Компилятор Microsoft Visual C++ практически не поддерживает никаких особенностей ревизий C99 и C11. В табл. E.2 перечислены компиляторы C, реализующие некоторые положения ревизии C11.

Таблица E.2 | Компиляторы, совместимые с ревизией C11

Компилятор	URL
GNU GCC	`cc.gnu.org/gcc-4.7/changes.html`
LLVM	`clang.llvm.org/docs/ReleaseNotes.html#cchanges`
IBM XL C	`www-01.ibm.com/software/awdtools/xlcpp/`
Pelles C	`www.smorgasbordet.com/pellesc/`

Текст предварительного, рабочего варианта стандарта можно найти по адресу: `www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`.

А окончательный вариант – по адресу: `webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2FISO%2FIEC+9899-2012`.

E.13.1 Новые заголовочные файлы в C11

В табл. E.3 перечислены новые заголовочные файлы, определяемые ревизией C11 стандарта.

Таблица E.3 | Стандартные заголовочные файлы, добавленные в ревизии C11

Стандартные заголовочные файлы	Описание
`<stdalign.h>`	Определяет типы для управления выравниванием в памяти
`<stdatomic.h>`	Определяет атомарные операции с объектами, используемые в многопоточном окружении
`<stdnoreturn.h>`	Функции, не возвращающие значений
`<threads.h>`	Библиотека управления потоками выполнения
`<uchar.h>`	Утилиты для работы с символами в кодировках UTF-16 и UTF-32

E.13.2 Поддержка многопоточной модели выполнения

Поддержка многопоточной модели выполнения – одно из важнейших улучшений в стандарте C11. Несмотря на то что модель многопоточного выполнения существует уже несколько десятилетий, интерес к ней быстро растет благодаря быстрому развитию многопроцессорных систем – даже смартфоны и планшетные компьютеры теперь снабжаются многоядерными процессорами. Наиболее распространенными в настоящее время являются двухъядерные процессоры, однако все большее распространение получают четырехъядерные процессоры. И количество ядер в процессорах продолжает увеличиваться. Многопроцессорные системы позволяют выполнять разные части одной задачи на разных процессорах, обеспечивая более высокую скорость решения этих задач. Чтобы извлечь максимальную выгоду из многопроцессорной архитектуры, необходимо писать многопоточные приложения. Когда программа разбивает задачи на отдельные потоки выполнения, многопроцессорная система может решать их параллельно.

Стандартная реализация многопоточной модели

Прежде библиотеки реализации многопоточной модели не были стандартизованы и являлись аппаратно-зависимыми расширениями языка C. Однако программистам хотелось бы писать код, легко переносимый между платформами. В этом заключается главное преимущество стандартизации. Заголовочный файл `<threads.h>`, определяемый ревизией C11, объявляет новые (необязательные) средства поддержки многопоточного выполнения, позволяющие писать переносимый код. На момент написания этих строк лишь немногие компиляторы обеспечивали поддержку стандартной многопоточной модели. При разработке примеров для этой главы мы использовали компилятор Pelles C (поддерживает только ОС Windows), который можно загрузить по адресу: `www.smorgasbordet.com/pellesc/`. Ниже мы представим основные механизмы поддержки многопоточного выполнения, позволяю-

щие создавать и запускать потоки. В конце этого раздела мы рассмотрим несколько других особенностей, связанных с поддержкой многопоточной модели, определяемых ревизией C11.

Запуск многопоточных программ

Когда какая-либо программа запускается, она начинает состязаться за обладание процессором (или процессорами) с операционной системой, с другими программами и с другими задачами, выполняемыми от имени операционной системы. Все виды задач обычно выполняются в фоновом режиме. Когда вы будете опробовать примеры, представленные в этом разделе, имейте в виду, что время, необходимое на вычисления, может отличаться в зависимости от быстродействия процессора в вашем компьютере и от количества ядер в процессоре. Это мало чем отличается от поездки в супермаркет. Время, затрачиваемое на дорогу, может отличаться от плотности дорожного движения, от погодных условий и многих других факторов. Иногда дорога может занять 10 минут, а иногда, особенно в часы пик, намного больше. То же справедливо и для компьютерных систем.

Кроме того, сказываются также накладные расходы, связанные с самим механизмом поддержки многопоточного выполнения. Простое деление задачи на два потока в двухъядерной системе не увеличит скорости выполнения в два раза, хотя определенный прирост все-таки будет наблюдаться. Как будет показано далее, многопоточные приложения на одноядерном процессоре показывают ухудшение производительности, по сравнению с последовательным, однопоточным выполнением операций.

Обзор примеров в этом разделе

Чтобы наглядно продемонстрировать преимущества многопоточных программ в многопроцессорных системах, в этом разделе будут представлены две программы:

- первая выполняет две вычислительные задачи последовательно;
- другая выполняет те же вычисления в параллельных потоках.

Мы выполнили обе программы на двух компьютерах – одно- и двухъядерном, работающих под управлением Windows 7, чтобы показать, как изменяется производительность программ в каждом случае. Мы определили время решения каждой задачи и общее время вычислений в обеих программах. В выводе программ наглядно видно увеличение производительности многопоточного приложения при выполнении в многоядерной системе.

Пример: последовательное выполнение двух вычислительных задач

В примере E.11 определена рекурсивная функция `fibonacci` (строки 37–46), которая была представлена в разделе 5.15. Напомним, что для вычисления больших чисел Фибоначчи рекурсивная реализация может потребовать много времени. Пример последовательно выполняет вычисления `fibonacci(50)`

(строка 16) и `fibonacci(49)` (строка 25). До и после каждого вызова функции `fibonacci` мы сохраняем текущее время, чтобы впоследствии вычислить время, затраченное на решение каждой задачи. Эти же значения используются для определения общего времени вычислений. В строках 21, 30 и 33 вызывается функция `difftime` (определена в заголовочном файле `<time.h>`), возвращающая количество секунд между двумя замерами времени.

Вывод программы разбит на три блока. В первом из них представлены результаты, полученные в двухъядерной системе, которые практически не отличаются от запуска к запуску. Во втором и третьем блоках даны результаты, полученные в одноядерной системе, которые могут варьировать, но всегда хуже, потому что программе приходится состязаться за обладание единственным процессором с другими программами и самой операционной системой.

Пример E.11 | Последовательное вычисление двух больших чисел Фибоначчи

```
 1 // Пример E.11: fibonacci.c
 2 // Последовательное вычисление двух больших чисел Фибоначчи
 3 #include <stdio.h>
 4 #include <time.h>
 5
 6 unsigned long long int fibonacci( unsigned int n ); // прототип функции
 7
 8 // выполнение программы начинается с функции main
 9 int main( void )
10 {
11    puts( "Sequential calls to fibonacci(50) and fibonacci(49)" );
12
13    // вычислить два числа Фибоначчи
14    time_t startTime1 = time( NULL );
15    puts( "Calculating fibonacci( 50 )" );
16    unsigned long long int result1 = fibonacci( 50 );
17    time_t endTime1 = time( NULL );
18
19    printf( "fibonacci( %u ) = %llu\n", 50, result1 );
20    printf( "Calculation time = %f minutes\n\n",
21       difftime( endTime1, startTime1 ) / 60.0 );
22
23    time_t startTime2 = time( NULL );
24    puts( "Calculating fibonacci( 49 )" );
25    unsigned long long int result2 = fibonacci( 49 );
26    time_t endTime2 = time( NULL );
27
28    printf( "fibonacci( %u ) = %llu\n", 49, result2 );
29    printf( "Calculation time = %f minutes\n\n",
30       difftime( endTime2, startTime2 ) / 60.0 );
31
32    printf( "Total calculation time = %f minutes\n",
33       difftime( endTime2, startTime1 ) / 60.0 );
34 } // конец main
35
36 // Рекурсивная функция вычисления чисел Фибоначчи
37 unsigned long long int fibonacci( unsigned int n )
38 {
```

```
39    // базовый случай
40    if ( 0 == n || 1 == n ) {
41       return n;
42    } // конец if
43    else { // шаг рекурсии
44       return fibonacci( n - 1 ) + fibonacci( n - 2 );
45    } // конец else
46 } // конец функции fibonacci
```

а) Результат в двухъядерной системе

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci( 50 )
fibonacci( 50 ) = 12586269025
Calculation time = 1.550000 minutes

Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 0.966667 minutes

Total calculation time = 2.516667 minutes
```

б) Результат в одноядерной системе

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci( 50 )
fibonacci( 50 ) = 12586269025
Calculation time = 1.600000 minutes

Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 0.950000 minutes

Total calculation time = 2.550000 minutes
```

с) Результат в одноядерной системе

```
Sequential calls to fibonacci(50) and fibonacci(49)
Calculating fibonacci( 50 )
fibonacci( 50 ) = 12586269025
Calculation time = 1.550000 minutes

Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 1.200000 minutes

Total calculation time = 2.750000 minutes
```

Пример: параллельное выполнение двух вычислительных задач

В примере E.12 также используется рекурсивная функция `fibonacci`, но на этот раз каждый вызов `fibonacci` выполняется в отдельном потоке. Первые два блока вывода получены при выполнении многопоточной версии программы в системе с двухъядерным процессором. Несмотря на вариации времени

выполнения, общее время вычисления чисел Фибоначчи (в наших тестах) всегда оказывалось лучше, чем время выполнения последовательной версии, представленной в примере E.11. Последние два блока вывода получены в системе с одноядерным процессором. И снова время выполнения не остается постоянным, но общее время выполнения всегда хуже, чем время выполнения последовательной версии из-за накладных расходов на конкуренцию с другими программами и между потоками за обладание процессором.

Пример E.12 | Параллельное вычисление двух больших чисел Фибоначчи

```
// Пример E.12: ThreadedFibonacci.c
// Параллельное вычисление двух больших чисел Фибоначчи
#include <stdio.h>
#include <threads.h>
#include <time.h>

#define NUMBER_OF_THREADS 2

int startFibonacci( void *nPtr );
unsigned long long int fibonacci( unsigned int n );

typedef struct ThreadData {
   time_t startTime;    // время запуска потока
   time_t endTime;      // время завершения потока
   unsigned int number; // порядковый номер числа Фибоначчи
} ThreadData; // конец структуры ThreadData

int main( void )
{
   // данные для передачи в потоки; используются назначенные инициализаторы
   ThreadData data[ NUMBER_OF_THREADS ] =
      { [ 0 ] = { .number = 50 },
        [ 1 ] = { .number = 49 } };

   // каждый поток должен иметь идентификатор типа thrd_t
   thrd_t threads[ NUMBER_OF_THREADS ];

   puts( "fibonacci(50) and fibonacci(49) in separate threads" );

   // создать и запустить потоки
   for ( unsigned int i = 0; i < NUMBER_OF_THREADS; ++i ) {
      printf( "Starting thread to calculate fibonacci( %d )\n",
         data[ i ].number );

      // создать поток и убедиться в успехе этой операции
      if ( thrd_create( &threads[ i ], startFibonacci, &data[ i ] ) !=
           thrd_success ) {

         puts( "Failed to create thread" );
      } // конец if
   } // конец for

   // ожидать завершения вычислений
   for ( int i = 0; i < NUMBER_OF_THREADS; ++i )
      thrd_join( threads[ i ], NULL );
```

```
46
47    // определить время запуска первого потока
48    time_t startTime = ( data[ 0 ].startTime > data[ 1 ].startTime ) ?
49       data[ 0 ].startTime : data[ 1 ].startTime;
50
51    // определить время завершения последнего потока
52    time_t endTime = ( data[ 0 ].endTime > data[ 1 ].endTime ) ?
53       data[ 0 ].endTime : data[ 1 ].endTime;
54
55    // вывести общее время вычислений
56    printf( "Total calculation time = %f minutes\n",
57       difftime( endTime, startTime ) / 60.0 );
58 } // конец main
59
60 // Вызывается потоком для рекурсивного вычисления числа Фибоначчи
61 int startFibonacci( void *ptr )
62 {
63    // привести указатель к типу ThreadData *,
64    // чтобы получить доступ к аргументам
65    ThreadData *dataPtr = (ThreadData *) ptr;
66    dataPtr->startTime = time( NULL ); // текущее время перед вычислениями
67
68    printf( "Calculating fibonacci( %d )\n", dataPtr->number );
69    printf( "fibonacci( %d ) = %lld\n",
70       dataPtr->number, fibonacci( dataPtr->number ) );
71
72    dataPtr->endTime = time( NULL );   // текущее время после вычислений
73
74    printf( "Calculation time = %f minutes\n\n",
75       difftime( dataPtr->endTime, dataPtr->startTime ) / 60.0 );
76    return thrd_success;
77 } // конец функции startFibonacci
78
79 // Рекурсивное вычисление числа Фибоначчи
80 unsigned long long int fibonacci( unsigned int n )
81 {
82    // базовый случай
83    if ( 0 == n || 1 == n ) {
84       return n;
85    } // конец if
86    else { // шаг рекурсии
87       return fibonacci( n - 1 ) + fibonacci( n - 2 );
88    } // конец else
89 } // конец функции fibonacci
```

а) Результат в двухъядерной системе

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci( 50 )
Starting thread to calculate fibonacci( 49 )
Calculating fibonacci( 50 )
Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 1.066667 minutes

fibonacci( 50 ) = 12586269025
Calculation time = 1.700000 minutes

Total calculation time = 1.700000 minutes
```

б) Результат в двухъядерной системе

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci( 50 )
Starting thread to calculate fibonacci( 49 )
Calculating fibonacci( 50 )
Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 0.683333 minutes

fibonacci( 50 ) = 12586269025
Calculation time = 1.666667 minutes

Total calculation time = 1.666667 minutes
```

в) Результат в одноядерной системе

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci( 50 )
Starting thread to calculate fibonacci( 49 )
Calculating fibonacci( 50 )
Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 2.116667 minutes

fibonacci( 50 ) = 12586269025
Calculation time = 2.766667 minutes

Total calculation time = 2.766667 minutes
```

г) Результат в одноядерной системе

```
fibonacci(50) and fibonacci(49) in separate threads
Starting thread to calculate fibonacci( 50 )
Starting thread to calculate fibonacci( 49 )
Calculating fibonacci( 50 )
Calculating fibonacci( 49 )
fibonacci( 49 ) = 7778742049
Calculation time = 2.233333 minutes

fibonacci( 50 ) = 12586269025
Calculation time = 2.950000 minutes

Total calculation time = 2.950000 minutes
```

Структура ThreadData

Функция, которую в данном примере выполняет каждый поток, принимает объект `ThreadData` в виде аргумента. Этот объект содержит число для передачи функции `fibonacci` и два члена типа `time_t`, где сохраняется текущее время до и после вызова `fibonacci` в каждом потоке. В строках 21–23 создается массив с двумя объектами `ThreadData`, члены `number` которых инициализируются с помощью назначенных инициализаторов значениями 50 и 49 соответственно.

Тип thrd_t

В строке 26 создается массив объектов типа `thrd_t`. При создании нового потока выполнения библиотека генерирует идентификатор потока и сохраняет его в объекте типа `thrd_t`. Идентификатор потока может затем использоваться с различными функциями управления потоками.

Создание и запуск потока

В строках 31–41 создаются два потока выполнения вызовом функции `thrd_create` (строка 36). Эта функция принимает три аргумента:

- указатель на объект типа `thrd_t`, где функция `thrd_create` должна сохранить идентификатор потока;
- указатель на функцию, выполняемую потоком (в данном случае `startFibonacci`), которая должна возвращать значение типа `int` и принимать указатель типа `void`, представляющий аргумент функции (в данном случае указатель на объект типа `ThreadData`); возвращаемое значение типа `int` определяет код завершения потока (например, `thrd_success` или `thrd_error`);
- указатель типа `void` на аргумент, который требуется передать функции, указанной во втором аргументе.

Функция `thrd_create` возвращает `thrd_success`, если поток был успешно создан, `thrd_nomem` – если для создания потока недостаточно памяти, и `thrd_error` в противном случае. Если поток был успешно создан, в контексте этого потока вызывается функция, указанная во втором аргументе.

Присоединение к потокам

Чтобы гарантировать, что программа не завершится до завершения всех ее потоков, в строках 44–45 для каждого созданного потока выполнения вызывается функция `thrd_join`. Это заставит программу ждать завершения потоков, и только после этого она сможет продолжить выполнять оставшийся код в функции `main`. Функция `thrd_join` принимает значение типа `thrd_t`, представляющее идентификатор потока для присоединения, и указатель на значение типа `int`, куда `thrd_join` сможет сохранить код завершения потока.

Функция startFibonacci

Функция `startFibonacci` (строки 61–77) определяет задачу, которую должны решать потоки, – в данном случае она вызывает функцию `fibonacci`, вычисляющую число Фибоначчи, запоминает время начала и конца вычислений, выводит результаты вычислений и время, потребовавшееся на эти вычисления (по аналогии с примером E.11). Поток продолжает выполняться, пока `startFibonacci` не вернет код завершения потока (`thrd_success`; строка 76), после чего он завершается.

Другие особенности поддержки многопоточного выполнения в C11
Помимо поддержки базовых возможностей многопоточной модели выполнения, представленных в этом разделе, ревизия C11 также включает другие особенности, такие как переменные `_Atomic` и атомарные операции, локальные переменные потоков, условные переменные и мьютексы. За дополнительной информацией по этим темам обращайтесь к разделам 6.7.2.4, 6.7.3, 7.17 и 7.26 стандарта, а также к следующим статьям:

```
blog.smartbear.com/software-quality/bid/173187/
     C11-A-New-C-Standard-Aiming-at-Safer-Programming
```

```
lwn.net/Articles/508220/
```

E.13.3 Функция quick_exit

Помимо функций `exit` (раздел 14.6) и `abort`, ревизия C11 определяет также функцию `quick_exit`, позволяющую завершить программу, – все три функции объявляются в заголовочном файле `<stdlib.h>`. Функция `quick_exit`, как и функция `exit`, принимает *код завершения* программы – обычно `EXIT_SUCCESS` или `EXIT_FAILURE`, но в некоторых платформах могут существовать и другие значения. Указанный код завершения возвращается программой вызывающему окружению, чтобы сообщить об успехе или неудаче. Функция `quick_exit` может, в свою очередь, вызвать до 32 других функций, выполняющих заключительные операции. Зарегистрировать эти функции можно с помощью `at_quick_exit` (похожей на функцию `atexit`, описанную в разделе 14.6). Они будут вызывать в порядке, *обратном* порядку их регистрации. Каждая зарегистрированная функция не должна принимать аргументы и ничего не должна возвращать. Причины введения функций `quick_exit` и `at_quick_exit` описываются в документе `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1327.htm`.

E.13.4 Поддержка Unicode®

Интернационализация и *локализация* – это процесс добавления в программное обеспечение поддержки *нескольких языков человеческого общения* и удовлетворения *национальных требований*, таких как форматы отображения денежных сумм. Набор символов Юникода (Unicode®) содержит символы множества алфавитов почти для всех языков, существующих в мире.

Ревизия C11 включает поддержку обеих кодировок – 16-битной (UTF-16) и 32-битной (UTF-32) символов Юникода, чем существенно упрощает интернационализацию и локализацию приложений. В разделе 6.4.5 стандарта C11 обсуждается, как создавать строковые литералы с символами Юникода. Раздел 7.28 стандарта описывает особенности заголовочного файла `<uchar.h>` с определениями новых утилит поддержки Юникода, включающего объявления новых типов `char16_t` и `char32_t` для поддержки кодировок UTF-16 и

UTF-32 соответственно. На момент написания этих строк новые особенности Юникода еще не получили широкой поддержки в компиляторах языка C.

E.13.5 Спецификатор функций _Noreturn

Спецификатор `_Noreturn` указывает, что функция не возвращает управление вызывающей программе. Например, функция `exit` (раздел 14.6) завершает программу, поэтому она не возвращает управление. В настоящее время все такие функции в стандартной библиотеке объявлены со спецификатором `_Noreturn`. Например, согласно стандарту C11, прототип функции `exit` выглядит так:

```
_Noreturn void exit(int status);
```

Если компилятору известно, что функция *не* возвращает управление, он сможет применить различные *оптимизации*. Он сможет также сгенерировать сообщение об ошибке, если в функции со спецификатором `_Noreturn` по неосторожности использована инструкция `return`.

E.13.6 Выражения обобщенного типа

Ревизия C11 определяет новое ключевое слово `_Generic`, реализующее механизм, который можно использовать для создания макросов (глава 13), вызывающих разные функции, в зависимости от типа аргумента. В C11 это ключевое слово теперь используется для определения обобщенных макросов в заголовочном файле `<tgmath.h>`. Многие математические функции имеют несколько версий, принимающих аргументы типов `float`, `double` или `long double`. Для вызова таких функций можно написать макрос, который автоматически будет вызывать функцию, соответствующую типу аргумента. Например, макрос `ceil` вызывает функцию `ceilf`, если его аргумент имеет тип `float`, функцию `ceil`, если аргумент имеет тип `double`, и функцию `ceill`, если аргумент имеет тип `long double`. Создание макросов с применением ключевого слова `_Generic` является слишком сложной темой, поэтому мы не будем рассматривать ее в данной книге. Особенности применения ключевого слова `_Generic` обсуждаются в разделе 6.5.1.1 стандарта C11.

E.13.7 Annex L: анализируемость и неопределенное поведение

Ревизия C11 стандарта определяет особенности языка, которые должны реализовать разработчики компиляторов. Из-за огромного разнообразия аппаратных и программных платформ (и других проблем) существует множество ситуаций, когда стандарт утверждает, что результат той или иной операции *не определен*. Такая неопределенность может повлечь за собой проблемы, связанные с безопасностью и надежностью программного обеспечения, – каждая операция, результат которой не определен, может оставить

систему открытой для нападений или способствовать появлению ошибок. Мы выполнили поиск фразы «не определен» («undefined behavior») в тексте стандарта C11 и нашли около 50 вхождений.

Члены группы CERT (cert.org), участвовавшие в разработке приложения Annex L к стандарту C11, тщательно исследовали ситуации, ведущие к неопределенному поведению, и обнаружили, что все они делятся на две основные категории: ситуации, когда разработчики компиляторов должны были сами предусмотреть оправданную реакцию, чтобы избежать серьезных последствий (эта категория называется «ограниченно неопределенные ситуации»), и ситуации, когда разработчики не в состоянии предложить что-либо разумное (эта категория называется «критически неопределенные ситуации»). Оказалось, что большинство ситуаций неопределенного поведения относятся к первой категории. Дэвид Китон (David Keaton) (исследователь подразделения CERT Secure Coding Program) описывает вышеупомянутые категории в следующей статье:

```
blog.sei.cmu.edu/post.cfm/improving-security-in-the-latest-cprogramming-
     language-standard-1.
```

Приложение Annex L к стандарту C11 определяет критически неопределенные ситуации. Добавление такого приложения в стандарт дает разработчикам компиляторов (совместимых с приложением Annex L) возможность реализовать достаточно разумную реакцию для большинства неопределенных ситуаций, которые могли игнорироваться в прежних реализациях. Приложение Annex L все еще не гарантирует оправданного поведения в критически неопределенных ситуациях. Разработчики программ могут определить, является ли компилятор совместимым с приложением Annex L, используя директивы условной компиляции (раздел 13.5), проверяющие доступность макроса __STDC_ANALYZABLE__.

E.13.8 Анонимные структуры и объединения

В главе 10 были представлены структуры и объединения. Ревизия C11 стандарта теперь поддерживает анонимные структуры и объединения, которые могут вкладываться в именованные структуры и объединения. Члены вложенной анонимной структуры или объединения интерпретируются как члены вмещающей структуры или объединения и доступны непосредственно, через вмещающий объект. Например, взгляните на следующее объявление структуры:

```
struct MyStruct {
   int member1;
   int member2;

   struct {
      int nestedMember1;
      int nestedMember2;
   }; // конец вложенной структуры
}; // конец вмещающей структуры
```

Если объявить переменную `myStruct` типа `struct MyStruct`, можно будет обращаться к членам структур, как показано ниже:

```
myStruct.member1;
myStruct.member2;
myStruct.nestedMember1;
myStruct.nestedMember2;
```

E.13.9 Управление выравниванием в памяти

В главе 10 обсуждался тот факт, что аппаратные платформы устанавливают определенные требования к выравниванию данных в памяти, в результате чего структуры занимают большие объемы памяти, чем суммарный размер их членов. Ревизия C11 позволяет определять собственные требования к выравниванию в памяти данных любых типов с помощью средств, объявленных в заголовочном файле `<stdalign.h>`. Спецификатор `_Alignas` применяется для определения требований к выравниванию. Оператор `alignof` возвращает требования к выравниванию для своего аргумента. Функция `aligned_alloc` позволяет динамически выделять память для объекта и определять требования к выравниванию в памяти. За дополнительной информацией обращайтесь к разделу 6.2.8 стандарта C11.

E.13.10 Статические утверждения

В разделе 13.10 мы познакомились с макросом `assert`, проверяющим значение выражения на этапе выполнения. Если результатом проверки является ложное значение, `assert` выводит сообщение об ошибке и вызывает функцию `abort`, чтобы завершить программу. Этот макрос удобно использовать для нужд отладки. Ревизия C11 стандарта определяет макрос `_Static_assert` для выполнения проверок константных выражений после обработки препроцессором, в момент, когда типы выражений уже известны. За дополнительной информацией обращайтесь к разделу 6.7.10 стандарта C11.

E.13.11 Вещественные типы

Ревизия C11 обеспечивает совместимость стандарта языка C со стандартом IEC 60559 вещественной арифметики, однако эта совместимость является необязательной для реализации.

E.14 Веб-ресурсы

Ресурсы C99

`www.open-std.org/jtc1/sc22/wg14/`

Официальный сайт комитета по стандартизации языка C. Здесь можно найти отчеты о недостатках, рабочие документы, проекты и промежуточные наработки, пояснения к стандарту C99, контактную информацию и многое другое.

`blogs.msdn.com/vcblog/archive/2007/11/05/iso-c-standard-update.aspx`
Статья Арджана Бьянки (Arjun Bijanki), ведущего инженера по тестированию компилятора Visual C++. Обсуждает причины отсутствия поддержки положений ревизии C99 в Visual Studio.

`www.ibm.com/developerworks/linux/library/l-c99/index.html`
Статья «Open Source Development Using C99» Питера Сибаха (Peter Seebach). Обсуждает особенности поддержки C99 в Linux и BSD.

`www.informit.com/guides/content.aspx?g=cplusplus&seqNum=215`
Статья «A Tour of C99» Дэнни Калева (Danny Kalev). Описывает некоторые ключевые особенности в ревизии C99 стандарта.

Стандарт C11

`webstore.ansi.org/RecordDetail.aspx?sku=INCITS%2FISO%2FIEC+9899-2012`
Продажа ANSI-варианта стандарта C11.

`www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf`
Последние три рабочие версии стандарта C11, предшествовавшие официально утвержденной и опубликованной версии.

Что нового в C11

`en.wikipedia.org/wiki/C11_(C_standard_revision)`[1]
Страница в Википедии с описанием нововведений, появившихся в C11 после выхода C99.

`progopedia.com/dialect/c11/`
На этой странице можно найти краткий список новых особенностей, появившихся в C11.

`www.informit.com/articles/article.aspx?p=1843894`
Статья «The New Features of C11» Дэвида Чизналла (David Chisnall).

`www.drdobbs.com/cpp/c-finally-gets-a-new-standard/232800444`
Статья «C Finally Gets a New Standard» Томаса Плюма (Thomas Plum). Обсуждает вопросы параллельного программирования, ключевые слова, класс хранения `thread_local`, дополнительные потоки выполнения и многое другое.

`www.drdobbs.com/cpp/cs-new-ease-of-use-and-how-the-language/240001401`
Статья «C's New Ease of Use and How the Language Compares with C++» Томаса Плюма (Thomas Plum). Обсуждает некоторые новые особенности C11, которые совпадают с особенностями в языке C++, и некоторые ключевые слова в C11, отличные от ключевых слов в языке C++, но имеющие схожую функциональность.

`www.i-programmer.info/news/98-languages/3546-new-iso-c-standard-c1x.html`
Статья «New ISO C Standard – C11» Майка Джеймса (Mike James). Кратко обсуждаются некоторые новые особенности.

---

[1] На русском языке: `ru.wikipedia.org/wiki/C11`. – *Прим. перев.*

`www.drdobbs.com/cpp/the-new-c-standard-explored/232901670`
Статья «The New C Standard Explored» Томаса Плюма (Thomas Plum). Обсуждаются функции из приложения C11 Annex K, безопасность функции `fopen()`, исправления в `tmpnam`, уязвимость спецификатора формата `%n`, улучшения в системе безопасности и др.

`m.drdobbs.com/144530/show/871e182fd14035dc243e815651ffaa79&t=qkoob97760e69a0dopqejjech4`
Статья «C's New Ease of Use and How the Language Compares with C++». В число обсуждаемых тем входят: выравнивание в памяти, строки Юникода и константы, обобщенные макросы, совместимость с языком C++.

`www.sdtimes.com/link/36892`
Статья «The Thinking behind C11» Джона Бенито (John Benito), члена рабочей группы ISO по стандартизации языка программирования C. В статье обсуждаются руководящие принципы, использовавшиеся комитетом по стандартизации языка C при работе над новой ревизией C11 стандарта.

Улучшения в системе безопасности

`blog.smartbear.com/software-quality/bid/173187/C11-A-New-C-Standard-Aiming-at-Safer-Programming`
Статья «C11: A New C Standard Aiming at Safer Programming» Дэнни Калева (Danny Kalev). Обсуждаются проблемы ревизии C99 стандарта и ожидания от стандарта C11 в терминах улучшения безопасности.

`www.amazon.com/exec/obidos/ASIN/0321335724/deitelassociatin`
Книга «Secure Coding in C and C++» Роберта Сикорда (Robert Seacord), обсуждаются улучшения в системе безопасности, реализованные в библиотеке Annex K.

`blog.sei.cmu.edu/post.cfm/improving-security-in-the-latest-c-programminglanguage-standard-1`
Статья «Improving Security in the Latest C Programming Language Standard» Дэвида Китона (David Keaton), члена группы CERT Secure Coding Program в институте программотехники имени Карнеги Меллона. Обсуждаются приемы проверки границ и анализируемость.

`blog.sei.cmu.edu/post.cfm/helping-developers-address-security-with-the-cert-csecure-coding-standard`
Статья «Helping Developers Address Security with the CERT C Secure Coding Standard» Дэвида Китона (David Keaton). Описывает, как решались проблемы безопасности в языке C с течением времени и при использовании правил CERT C Secure Coding Rules.

Проверка границ

`www.securecoding.cert.org/confluence/display/seccode/ERR03-C.+Use+runtimeconstraint+handlers+when+calling+the+bounds-checking+interfaces`
Статья на сайте института программотехники имени Карнеги Меллона «ERR03-C. Use runtime-constraint handlers when calling the bounds-checking

interfaces» Дэвида Свободы (David Svoboda). Содержит примеры совместимого и несовместимого кодов.

Параллельные вычисления

`stackoverflow.com/questions/8876043/multi-threading-support-in-c11`
Обсуждение на форуме «Multi-Threading support in C11». Обсуждается улучшенная модель упорядоченного доступа к памяти в C11 в сравнении с C99.

`www.t-dose.org/2012/talks/multithreaded-programming-new-c11-and-c11-standards`
Презентация «Multithreaded Programming with the New C11 and C++11 Standards» Класса ван Генда (Klass van Gend). Представляет новые особенности стандартов C11 и C++11 и обсуждает уровень поддержки новых стандартов в gcc и clang.

`www.youtube.com/watch?v=UqTirRXe8vw`
Видеоролик «Multithreading Using Posix in C Language and Ubuntu» Ахмада Насера (Ahmad Naser).

`supertech.csail.mit.edu/cilk/lecture-2.pdf`
Лекция «Multithreaded Programming in Cilk» Чарльза Лейсерсона (Charles E. Leiserson).

`fileadmin.cs.lth.se/cs/Education/EDAN25/F06.pdf`
Презентация «Threads in the Next C Standard» Йонаса Скептстедта (Jonas Skeppstedt).

`www.youtube.com/watch?v=gRe6Zh2M3zs`
Видеоролик Класса ван Генда (Klaas van Gend) с обсуждением особенностей многопоточного программирования в свете новых стандартов C11 и C++11.

`www.experiencefestival.com/wp/videos/c11-concurrency-part-7/4zWbQRE3tWk`
Серия видеороликов о параллельном программировании в C11.

Поддержка в компиляторах

`www.ibm.com/developerworks/ru/library/r-support-iso-c11/`
Статья «Поддержка стандарта ISO C11 в компиляторах IBM XL C/C++: новые функциональные возможности, реализованные в фазе 1». Содержит обзор новых возможностей, поддерживаемых компилятором, включая инициализацию комплексных значений, статические утверждения и функции, не возвращающие управления.

F

Отладчик Visual Studio

В этом приложении вы научитесь:

- устанавливать точки останова в отладчике;
- пользоваться командой **Continue** для продолжения выполнения программы;
- пользоваться окном **Locals** для просмотра и изменения значений переменных;
- пользоваться окном **Watch** для вычисления выражений;
- пользоваться командами **Step Into**, **Step Out** и **Step Over** управления выполнением;
- пользоваться окном **Autos** для просмотра переменных, используемых в окружающих инструкциях.

F.1 Введение

В этом приложении демонстрируются ключевые возможности отладчика Visual Studio. В приложении G обсуждаются особенности и возможности отладчика GNU. Для опробования примеров в этом разделе установите среду разработки Visual Studio Express for Windows Desktop, которую можно загрузить со страницы `microsoft.com/express`.

F.2 Точки останова и команда Continue

Наше знакомство с отладчиком мы начнем с исследования возможности определять **точки останова**, которые являются специальными маркерами, устанавливаемыми на любые строки выполняемого кода. Когда программа достигает точки останова, ее выполнение приостанавливается, что позволяет программисту заняться исследованием значений переменных с целью выявить логические ошибки. Например, вы можете проверить значение переменной, хранящей результат вычислений, чтобы определить, насколько правильно были выполнены эти вычисления. Имейте в виду, что попытка установить точку останова в строке, не являющейся выполняемым кодом (например, в строке с комментарием), приведет к установке точки останова в следующей выполняемой строке в этой же функции.

Для иллюстрации возможностей отладчика мы воспользуемся программой в примере F.1, которая определяет наибольшее из трех целых чисел. Выполнение программы начинается с функции `main` (строки 8–20). Три целых числа вводятся с помощью функции `scanf` (строка 15). Далее целые числа передаются функции `maximum` (строка 19), которая определяет наибольшее из них. Найденное значение возвращается функцией с помощью инструкции `return` (строка 36). Затем найденное значение выводится на экран функцией `printf` (строка 19).

Пример F.1 | Поиск наибольшего из трех целых чисел

```
1 /* Пример F.1: figF_01.c
2 Поиск наибольшего из трех целых чисел */
3 #include <stdio.h>
4
5 int maximum( int x, int y, int z ); /* прототип функции */
6
7 /* выполнение программы начинается с функции main */
8 int main( void )
9 {
```

```
10    int number1; /* первое целое число */
11    int number2; /* второе целое число */
12    int number3; /* третье целое число */
13
14    printf("%s", "Enter three integers: " );
15    scanf( "%d%d%d", &number1, &number2, &number3 );
16
17    /* number1, number2 и number3 - аргументы
18       в вызове функции maximum */
19    printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20 } /* конец main */
21
22 /* Определение функции maximum */
23 /* x, y и z - параметры */
24 int maximum( int x, int y, int z )
25 {
26    int max = x; /* предполагается, что x - наибольшее число */
27
28    if ( y > max ) { /* если y больше max, сохранить y в max */
29       max = y;
30    } /* конец if */
31
32    if ( z > max ) { /* если z больше max, сохранить z в max */
33       max = z;
34    } /* конец if */
35
36    return max; /* max - наибольшее значение */
37 } /* конец функции maximum */
```

Создание проекта в Visual Studio 2012 Express for Windows Desktop

Ниже описываются шаги по созданию проекта, включающего код из примера F.1.

1. В Visual Studio 2012 Express for Windows Desktop выберите пункт меню **File** > **New Project...** (**Файл** > **Создать проект...**), чтобы вывести диалог **New Project** (**Создать проект**).
2. В списке **Installed Templates** (**Установленные шаблоны**) в разделе **Visual C++** выберите пункт **Win32** и в центральной области диалога выберите ярлык **Win32 Console Application** (**Консольное приложение Win32**).
3. В поле **Name:** (**Имя:**) введите имя проекта и укажите каталог для сохранения в поле **Location:** (**Местоположение:**), затем щелкните на кнопке **OK**.
4. В диалоге **Win32 Application Wizard** (**Мастер приложений Win32**) щелкните на кнопке **Next** > (**Далее** >).
5. В разделе **Application type:** (**Тип приложения:**) выберите пункт **Console application** (**Консольное приложение**) и в разделе **Additional options:** (**Дополнительные параметры:**) снимите флажки **Precompiled header** (**Предварительно скомпилированный заголовок**) и **Security Development Lifecycle (SDL)** (**Проверки жизненного цикла разработки безопасного ПО**

(SDL)), выберите пункт **Empty project** (**Пустой проект**) и щелкните на кнопке **Finish** (Готово).

6. В окне **Solution Explorer** (**Обозреватель решений**) щелкните правой кнопкой мыши на папке **Source Files** (**Файлы исходного кода**) в своем проекте и выберите пункт контекстного меню **Add** > **Existing Item...** (**Добавить** > **Существующий элемент...**), чтобы вывести диалог **Add Existing Item** (**Добавление существующего элемента**).
7. Найдите папку с файлом, содержащим код из примера F.1, выберите его и щелкните на кнопке **Add** (**Добавить**).

Переход в режим отладки и установка точек останова

Ниже описываются шаги и различные команды отладчика, которые требуется выполнить для установки точек останова и исследования значения переменной `number1`, объявленной в примере F.1.

1. ***Включение отладчика.*** Обычно отладчик включен по умолчанию. Если это не так, измените значение в раскрывающемся списке **Solution Configurations** (**Конфигурации решения**), находящемся в панели инструментов (рис. F.1). Для этого щелкните на кнопке со стрелкой вниз рядом со списком и выберите пункт **Debug** (**Отладка**).

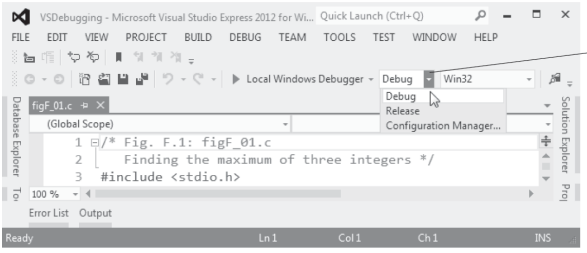


Рис. F.1 | Включение отладчика

2. ***Установка точек останова.*** Откройте файл `figF_01.c` двойным щелчком на нем в окне **Solution Explorer** (**Обозреватель решений**). Чтобы установить точку останова, щелкните внутри поля индикаторов (серое пространство слева от исходного текста программы, как показано на рис. F.2) напротив строки кода, где желательно остановить выполнение, или щелкните правой кнопкой мыши на строке кода и выберите пункт контекстного меню **Breakpoint** > **Insert Breakpoint** (**Точка останова** > **Вставить точку останова**). Количество одновременно устанавливаемых точек останова не ограничивается. Установите точки останова в строках 14 и 19. В поле индикаторов, в позиции щелчка мышью, появится красный кружок, свидетельствующий

о том, что точка останова установлена (рис. F.2). Если теперь запустить программу, отладчик будет приостанавливать ее выполнение в строках с точками останова. Про программу, приостановленную отладчиком, говорят, что она встала на точке останова. Точки останова можно устанавливать перед запуском программы, во время ее выполнения, а также когда она останавливается на точке останова.

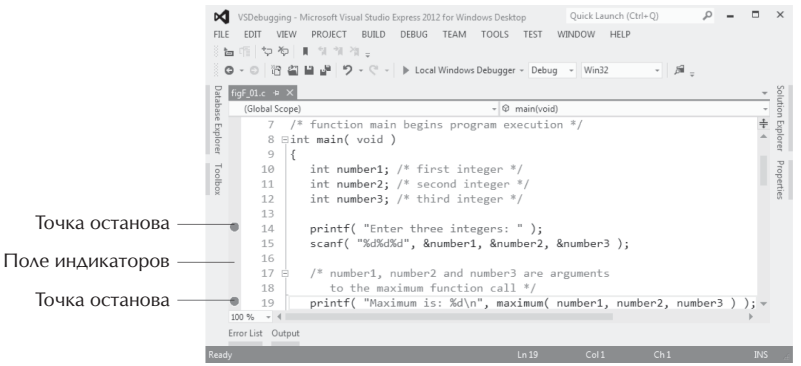


Рис. F.2 | Установка двух точек установа

3. ***Запуск в режиме отладки.*** После установки точек останова выберите пункт меню **Debug > Start Debugging** (**Отладка > Начать отладку**), чтобы скомпилировать программу и запустить процесс отладки. При отладке консольных приложений на экране появляется окно **Command Prompt** (**Командная строка**) (рис. F.3), где можно вводить данные, запрашиваемые программой, и наблюдать результаты. Достигнув точки останова в строке 14, отладчик приостановит выполнение программы.

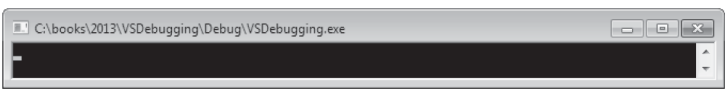


Рис. F.3 | Программа определения наибольшего числа запущена (снимок сделан до того, как она успела что-либо вывести)

4. ***Исследование результатов выполнения программы.*** После приостановки программы в строке 14 окно среды разработки активируется (рис. F.4). Желтая стрелка слева от строки 14 указывает, что эта строка является следующей, которая будет выполнена.

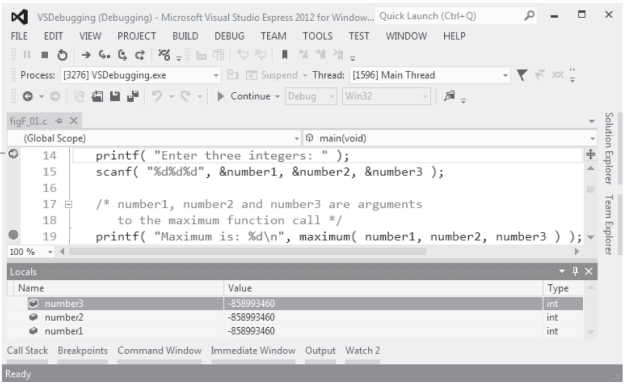


Рис. F.4 | Выполнение программы приостановлено на первой точке останова

5. ***Вызов команды Continue для возобновления выполнения.*** Чтобы возобновить выполнение программы, выберите пункт меню **Debug > Continue** (**Отладка > Продолжить**). После команды **Continue** программа продолжит выполнение до следующей точки останова или до конца функции `main`, смотря что встретится раньше. В данном случае программа продолжит выполнение и остановится для ввода чисел в строке 15. Введите значения 22, 85 и 17, разделив их пробелами. Программа продолжит выполнение дальше и остановится на следующей точке останова (строка 19). Если теперь поместить указатель мыши над именем переменной `number1`, во всплывающей подсказке отобразится ее значение (рис. F.5). Это может помочь в поиске логических ошибок в программах.
6. ***Установка точки останова на закрывающей фигурной скобке, завершающей определение функции `main`.*** Установите точку останова в строке 20, щелкнув напротив нее в поле индикаторов. Это не даст программе закрыться немедленно, сразу после вывода результатов. Если на пути потока выполнения нет других точек останова, программа завершит выполнение, и окно **Command Prompt** (**Командная строка**) закроется. Если не установить точку останова в строке 20, вы не успеете увидеть вывод программы до того, как окно закроется.
7. ***Возобновление выполнения программы.*** Воспользуйтесь командой **Debug > Continue** (**Отладка > Продолжить**), чтобы выполнить код до следующей точки останова. Программа выведет результаты вычислений, как показано на рис. F.6.
8. ***Удаление точки останова.*** Щелкните на точке останова внутри поля индикаторов.

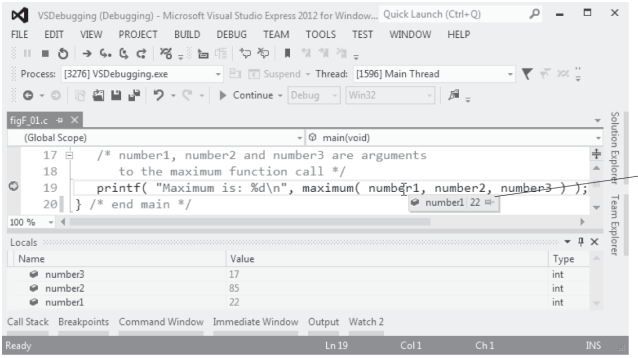


Рис. F.5 | Всплывающая подсказка со значением переменной

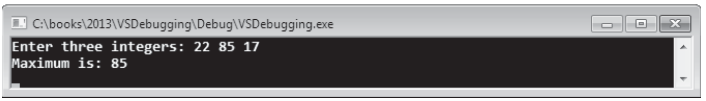


Рис. F.6 | Вывод программы

9. ***Завершение выполнения программы.*** Выберите пункт меню **Debug > Continue** (**Отладка > Продолжить**), чтобы завершить выполнение программы.

В этом разделе вы узнали, как включить отладчик и как устанавливать точки останова с целью исследования результатов вычислений. Вы также узнали, как возобновлять выполнение программы, приостановленной отладчиком в точке останова, и как удалять точки останова.

F.3 Окна Locals и Watch

В предыдущем разделе вы познакомились со всплывающими подсказками, позволяющими узнавать значения переменных. В этом разделе вы узнаете, как пользоваться окном **Locals** (**Локальные**), чтобы с его помощью изменять значения переменных в ходе выполнения программы. Вы также научитесь пользоваться окном **Watch** (**Контрольные значения**) для исследования значений более сложных выражений.

1. ***Добавление точек останова.*** Удалите имеющиеся точки останова щелчком мыши на каждой из них в поле индикаторов. Затем установите точку останова напротив строки 19.

2. ***Запуск сеанса отладки.*** Выберите пункт меню **Debug > Start** (**Отладка > Начать**). Введите значения 22, 85 и 17 в ответ на приглашение программы **Enter three integers:** и нажмите клавишу *Enter*.
3. ***Приостановка выполнения программы.*** Отладчик приостановит выполнение программы в строке 19. К этому моменту инструкция в строке 15 уже прочитает введенные вами значения и сохранит их в переменных `number1` (22), `number2` (85) и `number3` (17).
4. ***Исследование данных.*** После приостановки программы вы можете заняться исследованием значений своих локальных переменных в окне **Locals** (**Локальные**), которое во время отладки обычно отображается внизу слева в окне интегрированной среды разработки. Если его не видно, вывести окно **Locals** (**Локальные**) можно, выбрав пункт меню **Debug > Windows > Locals** (**Отладка > Окна > Локальные**). На рис. F.7 показаны значения локальных переменных `number1` (22), `number2` (85) и `number3` (17) в функции `main`.

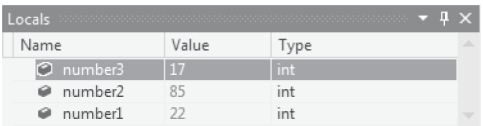


Рис. F.7 | Исследование переменных number1, number2 и number3

5. ***Вычисление арифметических и логических выражений.*** В окне **Watch** (**Контрольные значения**) можно вычислять арифметические и логические выражения. Выберите пункт меню **Debug > Windows > Watch 1** (**Отладка > Окна > Контрольные значения 1**). Введите в столбце **Name** (**Имя**) выражение `(number1 + 3) * 5`, Нажмите клавишу *Enter*. Значение этого выражения (125 в данном случае) появится в столбце **Value** (**Значение**), как показано на рис. F.9. В следующей строке, в столбце **Name** (**Имя**), введите выражение `number1 == 3` и нажмите клавишу *Enter*. Это выражение выясняет, равно ли значение переменной `number1` числу 3. Выражения, содержащие оператор `==` (или другие операторы отношения или равенства), интерпретируются как логические выражения. В данном случае в результате вычисления выражения получается значение `false` (рис. F.8), потому что в настоящий момент переменная `number1` содержит число 22, не равное 3.
6. ***Изменение значений.*** Исходя из значений, введенных пользователем (22, 85 и 17), наибольшим будет число 85. Однако с помощью окна **Locals** (**Локальные**) можно изменять значения переменных в ходе выполнения программы. Эта возможность может пригодиться

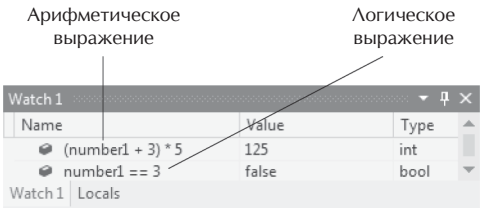


Рис. F.8 | Исследование значений выражений

для опытов с различными значениями и выявления логических ошибок. В окне **Locals** (**Локальные**) щелкните на поле **Value** (**Значение**) в строке с переменной `number1`. Введите число 90 и нажмите клавишу *Enter*. Отладчик изменит значение переменной `number1` и отобразит новое значение красным цветом (рис. F.9).

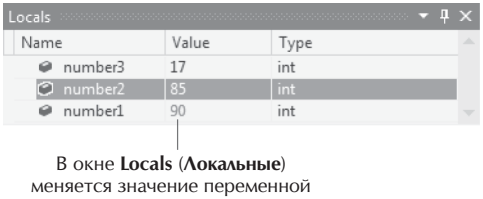


Рис. F.9 | Изменение значения переменной

7. ***Установка точки останова на закрывающей фигурной скобке, завершающей определение функции `main`.*** Установите точку останова в строке 20, чтобы не дать программе закрыться немедленно, сразу после вывода результатов. Если не установить эту точку останова, программа завершит выполнение, и вы не сможете увидеть вывод программы до того, как окно консоли закроется.
8. ***Возобновление выполнения и просмотр результатов работы программы.*** Выберите пункт меню **Debug > Continue** (**Отладка > Продолжить**), чтобы возобновить выполнение программы. Функция `main` продолжит выполнение до точки останова, и программа выведет результат, равный числу 90 (рис. F.10). Этот результат отражает тот факт, что на шаге 6 мы изменили значение переменной `number1`.
9. ***Остановка сеанса отладки.*** Выберите пункт меню **Debug > Stop Debugging** (**Отладка > Остановить отладку**). В результате окно **Command Prompt** (**Командная строка**) закроется. Удалите все точки останова.

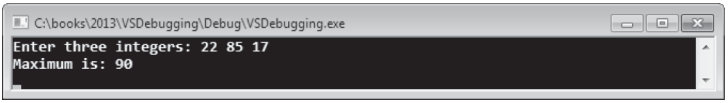


Рис. F.10 | Вывод программы после изменения значений переменной number1

В этом разделе вы узнали, как пользоваться окнами **Watch** (**Контрольные значения**) и **Locals** (**Локальные**) для вычисления арифметических и логических выражений. Вы также узнали, как изменить значение переменной в ходе выполнения программы.

F.4 Управление выполнением с помощью команд Step Into, Step Over, Step Out и Continue

Иногда пошаговое выполнение программы, строки за строкой, может помочь проверить правильность выполнения функций, а также обнаружить и исправить логические ошибки. Команды, с которыми вы познакомитесь в этом разделе, дают возможность построчно выполнить функцию, сразу все инструкции в функции или только оставшиеся инструкции в функции (если к этому моменту какие-то инструкции в функции уже были выполнены вами).

1. ***Установка точки останова.*** Установите точку останова в строке 19, щелкнув в поле индикаторов напротив строки с кодом.
2. ***Запуск отладчика.*** Выберите пункт меню **Debug** > **Start** (**Отладка** > **Начать**). Введите значения 22, 85 и 17 в ответ на приглашение программы к вводу **Enter three integers:** и нажмите клавишу *Enter*. По достижении точки останова в строке 19 выполнение программы приостановится.
3. ***Использование команды Step Into.*** Команда **Step Into** (**Шаг с заходом**) выполняет следующую инструкцию в программе (строка 19) и немедленно останавливается. Если строка содержит вызов функции (как в данном случае), управление будет передано вызванной функции. Это позволяет выполнить каждую инструкцию внутри функции в отдельности и исследовать особенности работы функции. Выберите пункт меню **Debug** > **Step Into** (**Отладка** > **Шаг с заходом**), чтобы войти в функцию `maximum`. Желтая стрелка остановится в строке 25, как показано на рис. F.11. Выберите пункт меню **Debug** > **Step Into** (**Отладка** > **Шаг с заходом**) еще раз, чтобы перейти к строке 26.
4. ***Использование команды Step Over.*** Выберите пункт меню **Debug** > **Step Over** (**Отладка** > **Шаг с обходом**), чтобы выполнить текущую ин-

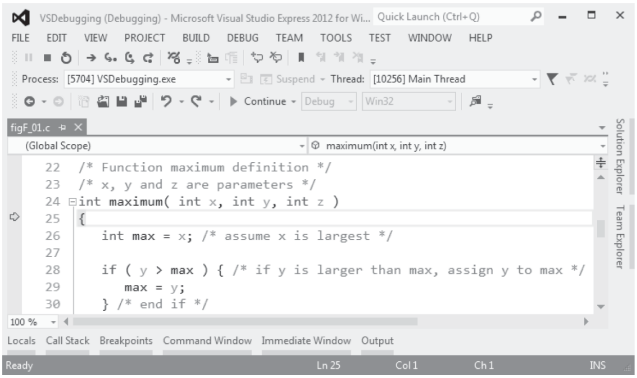


Рис. F.11 | Выполнение с заходом в функцию maximum

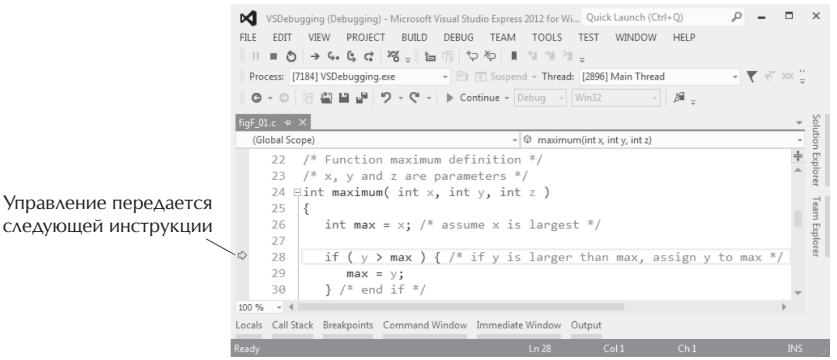


Рис. F.12 | Выполнение инструкций в функции maximum

струкцию (строка 26) и передать управление строке 28 (рис. F.12). Команда **Step Over** (**Шаг с обходом**) действует подобно команде **Step Into** (**Шаг с заходом**) при выполнении инструкции, не содержащей вызова функции. Отличия команды **Step Over** (**Шаг с обходом**) от команды **Step Into** (**Шаг с заходом**) станут более очевидны на шаге 10.

5. ***Использование команды Step Out.*** Выберите пункт меню **Debug > Step Out** (**Отладка > Шаг с выходом**), чтобы выполнить оставшиеся инструкции в функции и вернуть управление в точку, где функция была вызвана (строка 20 в примере F.1). Часто в длинных функциях бывает желательно посмотреть, как выполняется лишь несколько ключевых строк кода, а затем продолжить отладку в вызывающем коде. Коман-

да **Step Out** (**Шаг с выходом**) дает возможность продолжить выполнение программы в вызывающем коде без необходимости выполнять текущую функцию в пошаговом режиме.

6. ***Установка точки останова.*** Установите точку останова в конце функции `main`, в строке 20 примера F.1. Она будет использоваться в следующем шаге.
7. ***Использование команды Continue.*** Выберите пункт **Debug > Continue** (**Отладка > Продолжить**), чтобы продолжить выполнение программы до следующей точки останова в строке 20. Команду **Continue** (**Продолжить**) очень удобно использовать, когда требуется продолжить выполнение до следующей точки останова.
8. ***Остановка отладчика.*** Выберите пункт меню **Debug > Stop Debugging** (**Отладка > Остановить отладку**), чтобы завершить сеанс отладки. Эта команда закроет окно **Command Prompt** (**Командная строка**).
9. ***Запуск отладчика.*** Прежде чем продемонстрировать следующую особенность отладчика, необходимо снова начать сеанс отладки. Начните его, как описано в шаге 2, и введите числа 22, 85 и 17 в ответ на приглашение программы. Отладчик приостановит выполнение программы на точке останова в строке 19.
10. ***Использование команды Step Over.*** Выберите пункт меню **Debug > Step Over** (**Отладка > Шаг с обходом**). Напомню, что эта команда действует подобно команде **Step Into** (**Шаг с заходом**), если следующая выполняемая инструкция не содержит вызова функции. В противном случае она производит вызов функции и выполняет ее целиком (без захода в тело функции), а желтая стрелка перепрыгивает к следующей выполняемой строке (после вызова функции). В данном случае отладчик выполнит строку 19, где находится вызов функции `maximum`, получит результат ее вызова и отобразит его. Затем отладчик приостановит выполнение программы в строке 20 – на следующей выполняемой строке в текущей функции `main`.
11. ***Остановка отладчика.*** Выберите пункт меню **Debug > Stop Debugging** (**Отладка > Остановить отладку**). В результате окно **Command Prompt** (**Командная строка**) закроется. Удалите все точки останова.

В этом разделе вы узнали, как пользоваться командой отладчика **Step Into** (**Шаг с заходом**) для отладки функций, вызываемых в ходе выполнения программы. Вы увидели, как можно использовать команду **Step Over** (**Шаг с обходом**) для выполнения функций без захода в них. Вы опробовали команду **Step Out** (**Шаг с выходом**), позволяющую продолжить выполнение до конца текущей функции. Вы также познакомились с командой **Continue** (**Продолжить**), возобновляющей выполнение программы, пока не будет встречена очередная точка останова или до завершения программы.

F.5 Окно Autos

Окно **Autos** (**Видимые**) отображает переменные, использованные в предыдущей инструкции (включая возвращаемое значение функции, если таковое имеется), и переменные, используемые в следующей инструкции.

1. ***Установка точек останова.*** Установите точку останова в строке 19 щелчком мыши в поле индикаторов.
2. ***Использование окна Autos.*** Запустите отладчик, выбрав пункт меню **Debug > Start Debugging** (**Отладка > Начать отладку**), и введите числа 22, 85 и 17 в ответ на приглашение программы. Когда отладчик приостановит выполнение программы в точке останова, откройте окно **Autos** (**Видимые**), как показано на рис. F.13, выбрав пункт меню **Debug > Windows > Autos** (**Отладка > Окна > Видимые**). Так как вы только что ввели в программе три значения, в окне **Autos** (**Видимые**) окажутся три переменные: `number1`, `number2` и `number3` – с их новыми значениями. Отсортировав список переменных по их значениям, можно проверить правильность работы программы.

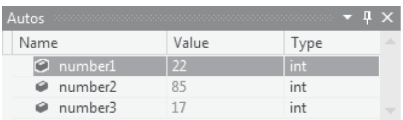


Рис. F.13 | Значения переменных `number1`, `number2` и `number3` в окне **Autos**

3. ***Ввод данных.*** Выберите пункт меню **Debug > Step Over** (**Отладка > Шаг с обходом**), чтобы выполнить строку 19. В окне **Autos** (**Видимые**) отобразятся возвращаемые значения обеих функций, `maximum` и `printf` (рис. F.14).

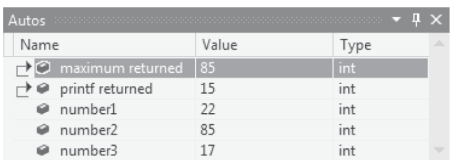


Рис. F.14 | Возвращаемые значения функций `maximum` и `printf` в окне **Autos**

4. ***Остановка отладчика.*** Выберите пункт меню **Debug > Stop Debugging** (**Отладка > Остановить отладку**), чтобы завершить сеанс отладки. Удалите все точки останова.

Отладчик GNU

В этом приложении вы научитесь пользоваться:

- командой `run` отладчика;
- командой `break` для установки точек останова;
- командой `continue` для возобновления выполнения программы;
- командой `print` для вычисления выражений;
- командой `set` для изменения значений переменных в ходе выполнения программы.

Краткое содержание

G.1 Введение

В главе 2 мы узнали, что существует два типа ошибок – ошибки компиляции и логические ошибки, и увидели, как исправлять ошибки компиляции. Логические ошибки не препятствуют компиляции программ, но они обусловливают получение ошибочных результатов. Операционная система GNU включает программное обеспечение, которое называется **отладчиком** и позволяет исследовать порядок выполнения программ с целью выявления и устранения логических ошибок.

Отладчик относится к разряду наиболее часто используемых инструментов разработки. Многие интегрированные среды разработки включают собственные отладчики, напоминающие отладчик GNU, или предоставляют графический интерфейс к отладчику GNU. В этом приложении мы познакомимся с ключевыми особенностями отладчика GNU. Дополнительные ссылки на руководства, описывающие приемы работы с отладчиками и другими инструментами разработки, можно найти на сайте нашего центра ресурсов C Resource Center (`www.deitel.com/c/`).

G.2 Точки останова и команды run, stop, continue и print

Наше исследование отладчика GNU мы начнем с **точек останова**, которые являются специальными маркерами, устанавливаемыми на любые строки выполняемого кода. Когда программа достигает точки останова, ее выполнение приостанавливается, что позволяет программисту заняться исследованием значений переменных с целью выявить логические ошибки. Например, вы можете проверить значение переменной, хранящей результат вычислений, чтобы определить, насколько правильно были выполнены эти вычисления. Имейте в виду, что попытка установить точку останова в строке, не являющейся выполняемым кодом (например, в строке с комментарием), приведет к установке точки останова в следующей выполняемой строке в этой же функции.

Для иллюстрации возможностей отладчика мы воспользуемся программой в примере G.1, которая определяет наибольшее из трех целых чисел. Выполнение программы начинается с функции `main` (строки 8–20). Три целых числа вводятся с помощью функции `scanf` (строка 15). Далее целые

числа передаются функции `maximum` (строка 19), которая определяет наибольшее из них. Найденное значение возвращается функцией с помощью инструкции `return` (строка 36). Затем найденное значение выводится на экран функцией `printf` (строка 19).

Пример G.1 | Поиск наибольшего из трех целых чисел

```
 1 /* Пример G.1: figG_01.c
 2 Поиск наибольшего из трех целых чисел */
 3 #include <stdio.h>
 4
 5 int maximum( int x, int y, int z ); /* прототип функции */
 6
 7 /* выполнение программы начинается с функции main */
 8 int main( void )
 9 {
10    int number1; /* первое целое число */
11    int number2; /* второе целое число */
12    int number3; /* третье целое число */
13
14    printf("%s", "Enter three integers: " );
15    scanf( "%d%d%d", &number1, &number2, &number3 );
16
17    /* number1, number2 и number3 - аргументы
18       в вызове функции maximum */
19    printf( "Maximum is: %d\n", maximum( number1, number2, number3 ) );
20 } /* конец main */
21
22 /* Определение функции maximum */
23 /* x, y и z - параметры */
24 int maximum( int x, int y, int z )
25 {
26    int max = x; /* предполагается, что x - наибольшее число */
27
28    if ( y > max ) { /* если y больше max, сохранить y в max */
29       max = y;
30    } /* конец if */
31
32    if ( z > max ) { /* если z больше max, сохранить z в max */
33       max = z;
34    } /* конец if */
35
36    return max; /* max - наибольшее значение */
37 } /* конец функции maximum */
```

Ниже описываются шаги и различные команды отладчика, которые требуется выполнить для установки точек останова и исследования значения переменной `number1`, объявленной в строке 10, в примере G.1.

1. ***Компиляция программы для отладки.*** Чтобы отладчик смог управлять программой, она должна быть скомпилирована с ключом `-g`, благодаря чему в выполняемый файл будет включена дополнительная информация, необходимая отладчику. Для этого введите команду

```
gcc -g figG_01.c
```

2. ***Запуск отладчика.*** Введите команду `gdb ./a.out` (пример G.2). Команда `gdb` запустит сеанс отладки и выведет приглашение к вводу (`gdb`), в котором можно вводить команды.
3. ***Запуск программы под управлением отладчика.*** Чтобы запустить программу под управлением отладчика, необходимо ввести команду `run` (пример G.3). Если перед запуском не установить точки останова в отладчике, программа будет выполнена целиком, от начала до конца, без остановок.

Пример G.2 | Запуск отладчика для отладки программы

```
$ gdb ./a.out
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.
html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...

(gdb)
```

Пример G.3 | Запуск программы без установки точек останова

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Enter three integers: 22 85 17
Max is 85

Program exited normally.
(gdb)
```

4. ***Установка точек останова в отладчике GNU.*** Установите точку останова в строке 14 в файле `figG_01.c` вводом команды `break 14`. Команда `break` вставит точку останова в строку с указанным номером. Количество одновременно устанавливаемых точек останова не ограничивается. Точки останова получают идентификаторы в соответствии с порядком, в котором они устанавливаются. Первая точка останова получит имя `Breakpoint 1`. Установите вторую точку останова в строке 19 вводом команды `break 19` (пример G.4). Эта новая точка останова получит имя `Breakpoint 2`. Если теперь запустить программу, отладчик будет приостанавливать ее выполнение в строках с точками останова. Точки останова можно устанавливать и после запуска программы под управлением отладчика. [*Обратите внима-*

ние: если у вас отсутствует листинг программы с пронумерованными строками, вы можете воспользоваться командой `list` для вывода строк кода с порядковыми номерами. Дополнительную информацию о команде `list` можно получить с помощью команды `help list`, которую следует вводить в строке приглашения `gdb`.]

Пример G.4 | Установка двух точек останова в программе

```
(gdb) break 14
Breakpoint 1 at 0x80483e5: file figG_01.c, line 14.
(gdb) break 19
Breakpoint 2 at 0x8048440: file figG_01.c, line 19.
```

5. ***Запуск программы и начало процесса отладки.*** Введите команду `run`, чтобы запустить программу и начать процесс отладки (пример G.5). Когда поток выполнения достигнет строки 14, отладчик приостановит программу. В этот момент отладчик известит вас, что встречена точка останова, и выведет исходный код в этой строке (14) – инструкцию, которая будет выполнена следующей.

Пример G.5 | Запуск программы и ее приостановка на первой точке останова

```
(gdb) run
Starting program: /home/user/AppJ/a.out

Breakpoint 1, main() at figG_01.c:14
14                    printf("%s", "Enter three integers: " );
(gdb)
```

6. ***Использование команды continue для возобновления выполнения программы.*** Введите команду `continue`. Эта команда возобновит выполнение программы, которое продолжится до следующей точки останова (строка 19). Введите числа 22, 85 и 17 в приглашении программы к вводу. Отладчик известит вас, когда будет достигнута вторая точка останова (пример G.6). Обратите внимание, что вывод программы `figG_01` появляется между сообщениями отладчика.

Пример G.6 | Продолжение выполнения программы до следующей точки останова

```
(gdb) continue
Continuing.
Enter three integers: 22 85 17

Breakpoint 2, main() at figG_01.c:19
19              printf("Maximum is %d\n", maximum(number1, number2,
number3));
( gdb)
```

7. ***Исследование значения переменной.*** Введите команду `print number1`, чтобы вывести текущее значение переменной `number1` (пример G.7). Команда `print` позволяет «заглянуть» в память компьютера и увидеть значения переменных. Такая возможность может пригодиться для поиска и устранения логических ошибок в коде. В данном случае в переменной хранится значение 22, введенное нами и присвоенное переменной `number1` в строке 15.

Пример G.7 | Вывод значений переменных

```
(gdb) print number1
$1 = 22
(gdb)
```

8. ***Использование вспомогательных переменных.*** При использовании команды `print` результаты сохраняются во вспомогательных переменных, таких как `$1`. Вспомогательные переменные – это временные переменные, создаваемые отладчиком. Им присваиваются имена, начинающиеся с символа доллара, за которым следует целое число. Вспомогательные переменные можно использовать для вычисления значений арифметических и логических выражений. Введите команду `print $1`, и отладчик выведет значение переменной `$1` (пример G.8), хранящей значение переменной `number1`. Обратите внимание, что попытка вывести значение переменной `$1` не приведет к созданию новой вспомогательной переменной `$2`.

Пример G.8 | Вывод значения вспомогательной переменной

```
(gdb) print $1
$2 = 22
(gdb)
```

9. ***Продолжение выполнения программы.*** Введите команду `continue`, чтобы продолжить выполнение программы. Поскольку далее нет никаких точек останова, отладчик выполнит программу до конца (пример G.9).

Пример G.9 | Завершение выполнения программы

```
(gdb) continue
Continuing
Max is 85

Program exited normally
(gdb)
```

10. ***Удаление точек останова.*** Вывести список точек останова, установленных в программе, можно командой `info break`. Удалить точ-

ку останова можно командой `delete`, передав ей номер удаляемой точки останова. Удалите первую точку останова командой `delete 1`. Удалите также вторую точку останова. Теперь введите команду `info break`, чтобы вывести список точек останова, оставшихся в программе. Отладчик должен ответить, что точек останова не осталось (пример G.10).

Пример G.10 | Обзор и удаление точек останова

```
(gdb) info break
Num  Type        Disp    Enb   Address       What
1    breakpoint  keep    y     0x080483e5    in main at figG_01.c:14
         breakpoint already hit 1 time
2    breakpoint  keep    y     0x08048799    in main at figG_01.c:19
         breakpoint already hit 1 time
(gdb) delete 1
(gdb) delete 2
(gdb) info break
No breakpoints or watchpoints
(gdb)
```

11. ***Выполнение программы без точек останова.*** Введите команду `run`, чтобы выполнить программу. Введите числа 22, 85 и 17 в ответ на приглашение программы. Так как две имевшиеся точки останова были удалены, программа будет выполнена без остановок в отладчике (пример G.11).
12. ***Использование команды quit.*** Введите команду `quit`, чтобы завершить сеанс работы с отладчиком (пример G.12). Эта команда вызовет завершение отладчика.

Пример G.11 | Выполнение программы без точек останова

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Enter three integers: 22 85 17
Max is 85

Program exited normally.
(gdb)
```

Пример G.12 | Завершение сеанса отладки с помощью команды quit

```
(gdb) quit
$
```

В этом разделе вы научились запускать отладчик командой `gdb` и использовать его команду `run` для запуска программы под управлением отладчика. Вы узнали, как устанавливать точки останова в строках с определенными

номерами, внутри функции `main`. Команду `break` можно также использовать для установки точек останова в других файлах или в определенной функции. Для этого достаточно передать команде `break` имя файла с номером строки через двоеточие или имя функции соответственно. В последнем случае отладчик будет останавливать выполнение программы сразу после входа потока выполнения в указанную функцию.

В этом разделе вы также познакомились с командой `help list`, позволяющей получить дополнительную информацию о команде `list`. Если у вас появятся какие-либо вопросы, касающиеся отладчика или его команд, введите команду `help` или `help` с именем интересующей команды.

Наконец, вы узнали, как можно получить значение переменной с помощью команды `print` и как удалять точки останова с помощью команды `delete`. Вы также познакомились с командой `continue`, возобновляющей выполнение программы после приостановки ее на точке останова, и с командой `quit`, завершающей отладчик.

G.3 Команды print и set

В предыдущем разделе вы узнали, как пользоваться командой `print` отладчика для получения значения переменной в ходе выполнения программы. В этом разделе вы узнаете, как с помощью той же команды `print` исследовать значения более сложных выражений. Вы также познакомитесь с командой `set`, позволяющей присваивать новые значения переменным. В этом разделе мы будем предполагать, что вы работаете в каталоге, где хранятся файлы с исходным кодом примера для этого приложения, и уже скомпилировали его с ключом `-g`.

1. ***Запуск отладчика.*** Введите команду `gdb /a.out`, чтобы запустить отладчик GNU.
2. ***Установите точку останова***. Установите точку останова в строке 19 вводом команды `break 19` (пример G.13).

Пример G.13 | Установка точки останова в программе

```
(gdb) break 19
Breakpoint 1 at 0x8048412: file figG_01.c, line 19.
(gdb)
```

3. ***Запуск программы и ее выполнение до точки останова.*** Введите команду `run`, чтобы запустить программу (пример G.14). Программа будет выполняться, пока не будет достигнута точка останова в строке 19. После этого выполнение программы будет приостановлено отладчиком. Инструкция в строке 19 – это инструкция, которая будет выполнена следующей.

Пример G.14 | Выполнение программы до точки останова в строке 19

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Enter three integers: 22 85 17

Breakpoint 1, main() at figG_01.c:19
19          printf("Maximum is %d\n", maximum(number1, number2, number3));
(gdb)
```

4. ***Вычисление арифметических и логических выражений.*** Как уже говорилось в разделе G.2, после приостановки программы вы можете заняться исследованием значений переменных в программе с помощью команды `print`. Однако команду `print` можно также использовать для вычисления арифметических и логических выражений. Введите команду `print number1 - 2`. В результате отладчик должен вывести число 20 (пример G.15), но при этом фактическое значение переменной `number1` не изменится. Введите команду `print number1 == 20`. Выражения, содержащие оператор `==`, возвращают 0, если условие ложно, и 1 – если условие истинно. Отладчик в данном случае вернет число 0 (пример G.15), потому переменная `number1` все еще хранит значение 22.

Пример G.15 | Вычисление выражений в отладчике

```
(gdb) print number1 - 2
$1 = 20
(gdb) print number1 == 20
$2 = 0
(gdb)
```

5. ***Изменение значения переменной.*** В ходе выполнения программы под управлением отладчика имеется возможность изменять значения переменных. Это может пригодиться для исследования поведения программы с разными значениями переменных и для поиска логических ошибок. Изменить значение переменной можно с помощью команды `set`. Введите команду `set number1 = 90`, чтобы изменить значение переменной `number1`, а затем введите команду `print number1`, чтобы убедиться, что переменной действительно было присвоено новое значение (пример G.16).

Пример G.16 | Изменение значения переменной после приостановки на точке останова

```
(gdb) set number1 = 90
(gdb) print number1
$3 = 90
(gdb)
```

6. ***Обзор результатов выполнения программы.*** Введите команду `continue`, чтобы продолжить выполнение программы. В результате будет выполнена строка 19 программы, которая вызовет функцию `maximum` и передаст ей числа `number1`, `number2` и `number3`. Затем функция `main` выведет наибольшее число. Обратите внимание, что программа вывела число 90 (пример G.17). Это обусловлено изменением значения переменной `number1` на предыдущем шаге.

Пример G.17 | Использование изменившейся переменной

```
(gdb) continue
Continuing.
Max is 90

Program exited normally.
(gdb)
```

7. ***Использование команды quit.*** Введите команду `quit`, чтобы завершить сеанс отладки (пример G.18). Эта команда завершит отладчик.

Пример G.18 | Завершение сеанса отладки с помощью команды quit

```
(gdb) quit
$
```

В этом разделе вы узнали, как с помощью команды `print` отладчика вычислять арифметические и логические выражения, а также познакомились с командой `set`, дающей возможность изменять значения переменных в ходе выполнения программы.

G.4 Управление выполнением с помощью команд step, finish и next

Иногда пошаговое выполнение программы, строки за строкой, может помочь проверить правильность выполнения функций, а также обнаружить и исправить логические ошибки. Команды, с которыми вы познакомитесь в этом разделе, дают возможность построчно выполнить функцию, сразу все инструкции в функции или только оставшиеся инструкции в функции (если к этому моменту какие-то инструкции в функции уже были выполнены вами).

1. ***Запуск отладчика.*** Запустите отладчик вводом команды `gdb ./a.out`.
2. ***Установка точек останова.*** Введите команду `break 19`, чтобы установить точку останова в строке 19.

3. ***Выполнение программы.*** Запустите программу командой `run`, затем введите числа 22, 85 и 17 в ответ на приглашение программы. После этого отладчик сообщит, что достигнута точка останова, и выведет код в строке 19. Отладчик приостановится и будет ждать ввода дальнейших команд.
4. ***Использование команды step.*** Команда `step` выполняет следующую инструкцию в программе. Если следующей инструкцией является вызов функции, управление будет передано в вызываемую функцию. Команда `step` позволяет входить в функции и исследовать выполнение отдельных инструкций в них. Например, для обзора и изменения переменных внутри функции можно использовать команды `print` и `set`. Введите команду `step`, чтобы войти в функцию `maximum` (в примере G.1). Отладчик сообщит, что команда `step` выполнена, и выведет следующую инструкцию (пример G.19) – в данном случае код в строке 28 из файла `figG_01.c`.

Пример G.19 | Использование команды step для входа в функцию

```
(gdb) step
maximum (x=22, y=85, z=17) at figG_01.c:26
28      int max = x;
(gdb)
```

5. ***Использование команды finish.*** После входа в функцию `maximum` введите команду `finish`. Эта команда выполнит оставшиеся инструкции в функции и вернет управление в точку ее вызова. После выхода из функции команда `finish` остановится в строке 19 (пример G.20). При этом отладчик выведет значение, которое вернула функция `maximum`. Часто в длинных функциях бывает желательно посмотреть, как выполняется лишь несколько ключевых строк кода, а затем продолжить отладку в вызывающем коде. Команда `finish` дает возможность продолжить выполнение программы в вызывающем коде без необходимости выполнять текущую функцию в пошаговом режиме.

Пример G.20 | Использование команды finish для выхода из функции

```
(gdb) finish
Run till exit from #0   maximum ( x = 22, y = 85, z = 17) at
figG_01.c:26
0x0804842b in main() at figG_01.c:19
19         printf("Maximum is %d\n", maximum(number1, number2, number3));
Value returned is $1 = 85
(gdb)
```

6. ***Использование команды continue для возобновления выполнения.*** Введите команду `continue`, чтобы продолжить выполнение программы до ее завершения.

7. ***Повторный запуск программы.*** Точки останова продолжают существовать до окончания сеанса отладки, в котором они были установлены. Поэтому точка останова, определенная в шаге 2, продолжает действовать. Введите команду `run`, чтобы запустить программу, и введите числа 22, 85 и 17 в ответ на ее приглашение. Как и в шаге 3, программа будет приостановлена отладчиком на точке останова в строке 19, после чего отладчик будет ждать от вас дальнейших команд (пример G.21).

Пример G.21 | Повторный запуск программы

```
(gdb) run
Starting program: /home/user/AppJ/a.out
Enter three integers: 22 85 17

Breakpoint 1, main() at figG_01.c:19
19         printf("Maximum is %d\n", maximum(number1, number2, number3));
(gdb)
```

8. ***Использование команды next.*** Введите команду `next`. Эта команда действует подобно команде `step`, если следующая выполняемая инструкция не содержит вызова функции. В противном случае она производит вызов функции, выполняет ее целиком, без захода в тело функции, и останавливается в следующей строке (пример G.22). В шаге 4 команда `step` выполняла вход в вызываемую функцию. В данном случае команда `next` выполнит функцию `maximum` и выведет наибольшее из трех целых чисел. После этого отладчик остановится в строке 20.

Пример G.22 | Использование команды next для выполнения функции без захода в нее

```
(gdb) next
Max is 85
20         }
(gdb)
```

9. ***Использование команды quit.*** Введите команду `quit`, чтобы завершить сеанс отладки (пример G.23). Если ввести эту команду, когда программа еще не завершилась, выполнение программы будет немедленно прервано и оставшиеся в ней инструкции не будут выполнены.

Пример G.23 | Завершение сеанса отладки командой qui

```
(gdb) quit
The program is running. Exit anyway? (y or n) y
$
```

В этом разделе вы узнали, как пользоваться командами отладчика `step` и `next` для отладки функций, вызываемых в ходе выполнения программы. Вы увидели, как можно использовать команду `next` для выполнения функций с заходом в них, а также познакомились с возможностью команды `quit` завершать сеанс отладки с одновременным прерыванием выполнения программы.

G.5 Команда watch

Команда `watch` предписывает отладчику начать слежение за указанной переменной. Когда значение переменной изменится, отладчик сообщит об этом. В данном разделе мы будем использовать команду `watch`, чтобы уловить момент, когда изменяется значение переменной `number1`.

1. ***Запуск отладчика.*** Запустите отладчик вводом команды `gdb ./a.out`.
2. ***Установка точек останова и запуск программы.*** Введите команду `break 14`, чтобы установить точку останова в строке 14. Затем запустите программу командой `run`. Отладчик остановит выполнение программы на точке останова в строке 14 (пример G.24).

Пример G.24 | Выполнение программы до первой точки останова

```
(gdb) break 14
Breakpoint 1 at 0x80483e5: file figG_01.c, line 14.
(gdb) run
Starting program: /home/user/AppJ/a.out

Breakpoint 1, main () at figG_01.c:14
14          printf("%s", "Enter three integers: " );
(gdb)
```

3. ***Включение слежения за переменной.*** Включите слежение за переменной `number1` вводом команды `watch number1` (пример G.25). Это задание будет помечено как `watchpoint 2`, потому что для нумерации заданий для слежения и точек останова выполняется в общей последовательности. Вы можете организовать слежение за любой переменной, находящейся в текущей области видимости. Как только значение переменной, находящейся под наблюдением, изменится, отладчик остановит выполнение программы и сообщит изменившееся значение.

Пример G.25 | Включение наблюдения за переменной

```
(gdb) watch number1
Hardware watchpoint2: number1
(gdb)
```

4. ***Продолжение выполнения программы.*** Введите команду `continue`, чтобы продолжить выполнение программы, и введите три числа в ответ на ее приглашение. Как только значение переменной `number1` изменится, отладчик немедленно сообщит об этом и приостановит программу (пример G.26). Старое значение (Old value) переменной `number1` – это значение, которое она имела до инициализации. Это значение может отличаться в разных прогонах программы. Данная непредсказуемость лишний раз доказывает важность инициализации всех переменных перед их использованием.

Пример G.26 | Приостановка программы после изменения значения переменной

```
(gdb) continue
Continuing.
Enter three integers: 22 85 17
Hardware watchpoint 2: number1

Old value = -1208401328
New value = 22
0xb7e6c692 in _IO_vfscanf() from /lib/i686/cmov/libc.so.6
(gdb)
```

5. ***Продолжение выполнения программы.*** Введите команду `continue` – программа завершит выполнение функции `main`. Отладчик автоматически удалит задание на слежение за переменной `number1`, потому что она вышла из области видимости. Удаление задания на слежение вынудит отладчик приостановить выполнение программы. Введите команду `continue` еще раз, чтобы завершить выполнение программы (пример G.27).

Пример G.27 | Продолжение выполнения программы до конца

```
(gdb) continue
Continuing.
Max is 85

Watchpoint 2 is deleted because the program has left the block in
which its expression is valid.
0xb7e4aab7 in exit() from /lib/i686/cmov/libc.so.6
(gdb) continue
Continuing

Program exited normally
(gdb)
```

6. ***Перезапуск отладчика и повторное включение слежения за переменной.*** Введите команду `run`, чтобы запустить программу. Снова включите слежение за переменной `number1` вводом команды `watch`

`number1`. На этот раз задание слежения будет помечено как `watchpoint 3`. Введите команду `continue`, чтобы продолжить выполнение (пример G.28).

7. ***Выключение слежения за переменной.*** Допустим, что вам требуется понаблюдать за изменением значения переменной только в пределах ограниченного фрагмента программы. Покинув этот фрагмент, вы можете выключить слежение за переменной `number1` вводом команды `delete 3` (пример G.29). Введите команду `continue` – программа завершит выполнение без остановок.

Пример G.28 | Повторное включение слежения

```
(gdb) run
Starting program: /home/users/AppJ/a.out

Breakpoint 1, main () at figG_01.c:14
14          printf("Enter three integers: ");
(gdb) watch number1
Hardware watchpoint 3: number1
(gdb) continue
Continuing
Hardware watchpoint 3: number1

Old value = -1208798640
New value = 22
0xb7e0b692 in _IO_vfscanf() from /lib/i686/cmov/libc.so.6
(gdb)
```

Пример G.29 | Выключение слежения

```
(gdb) delete 3
(gdb) continue
Continuing.
Max is 85

Program exited normally.
(gdb)
```

В этом разделе вы научились пользоваться командой `watch` для организации слежения за изменениями в переменных. Вы также узнали, как с помощью команды `delete` выключать слежение за переменными до окончания выполнения программы.

Алфавитный указатель

Символы

T

U

V

W

А

Б

В

Г

Д

Н

О

П

P

C

T

Ш

Э

Я

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС-КНИГА» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес **books@alians-kniga.ru**.

Дейтел Пол, Дейтел Харви

С для программистов с введением в С11

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 20.01.2014. Формат 60×90 1/16 .
Гарнитура «Гарамонд». Печать офсетная.
Усл. печ. л. 32. Тираж 200 экз.

Веб-сайт издательства: www.дмк.рф

C для программистов

Руководство по процедурному программированию, содержащее более 130 действующих примеров кода, от издательства Deitel® для профессиональных программистов

Пол Дейтел и Харви Дейтел *являются основателями компании Deitel & Associates, Inc., известной во всем мире своими корпоративными тренингами и услугами по организации разработки программного обеспечения. Миллионы людей по всему миру покупают книги, электронные книги, видеоуроки и другие ресурсы издательства Deitel, с целью овладеть C, C++, Visual C++®, Java™, C#, Visual Basic®, Android™, iOS®, веб-программированием, HTML5, JavaScript®, CSS3, XML, Perl, Python® и многими другими технологиями.*

На практических примерах рассматриваются следующие темы:

- основы программирования на C;
- компиляция и отладка с помощью GNU gcc и gdb, а также Visual C++®;
- ключевые новшества в ревизии C11 стандарта: обобщенные выражения, анонимные структуры и объединения, выравнивание в памяти, расширенная поддержка Unicode®, _Static_assert, quick_exit и at_quick_exit, спецификатор функций _Noreturn, новые заголовочные файлы;
- обновленная модель многопоточного программирования в C11, улучшающая производительность приложений в современных многоядерных системах;
- безопасное программирование на C;
- структуры данных, поиск и сортировка;
- проблемы, связанные с порядком вычислений, препроцессор;
- назначенные инициализаторы, составные литералы, тип bool, комплексные числа, массивы переменной длины, ограниченные указатели, обобщенные макросы математических операций, встраиваемые функции, и многое другое.

«Отличное введение в язык программирования C со множеством ясных примеров. Ловушки языка C четко обозначены и определены приемы программирования, позволяющие избежать их.»

Джон Бенито (John Benito), Blue Pilot Consulting, Inc., и член ISO WG14 – рабочей группы по стандартизации языка программирования C

«Обширные примеры с подробными описаниями в тексте делают эту книгу лучшей из тех, что я когда-либо видел. Возможность опробовать код примеров в сочетании с чтением текста позволяет читателям глубже понять особенности языка C.»

Том Ретард (Tom Rethard), Арлингтонский университет штата Техас

«Знакомит с программированием на языке C, своими советами и освещением наиболее удачных приемов помогает вам стать ценным специалистом на рынке труда. Хорошее описание проблем переносимости кода.»

Хемант Х. М. (Hemanth H.M.), программист в SonicWALL

ISBN 978-5-97060-073-3

PRENTICE HALL

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Internet-магазин:
www.dmkpress.com
Книга - почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
тел. (499)725-5409
e-mail: books@alians-kniga.ru