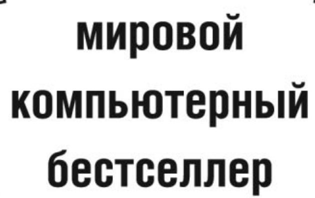
Грег Перри
Дин Миллер

Научись программировать быстро и качественно!

Программирование на C для начинающих

3-е издание

ЭКСМО


мировой
компьютерный
бестселлер

Greg Perry
Dean Miller

C Programming

Absolute begginer's guide

Third Edition

que®

Грег Перри
Дин Миллер

Программирование на C для начинающих

3-е издание

ЭКСМО
Москва
2015

УДК 004.43
ББК 32.973.26-018.1
П27

Gregg Perry, Dean Miller
C Programming
Absolute Beginner's Guide

Authorized translation for the English language edition, entitled C PROGRAMMING ABSOLUTE BEGINNER'S GUIDE, 3rd Edition; ISBN 0789751984 by PERRY, GREG; and MILLER, DEAN; published by Pearson Education, Inc, publishing as QUE Publishing. Copyright © 2014 by Pearson Education, Inc. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Все права защищены. Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения правообладателя Pearson Education, Inc.

В оформлении переплета использована фотография:
violetkaipa / Shutterstock.com
Используется по лицензии от Shutterstock.com

Перри, Грег.

П27 Программирование на C для начинающих / Грег Перри, Дин Миллер. — 3-е издание. — Москва : Эксмо, 2015. — 368 с. — (Мировой компьютерный бестселлер).

Простое и понятное руководство по программированию на C поможет быстро научиться программированию. Подробные объяснения и интересные примеры сделают процесс обучения легким. Вы легко освоите все основные функции C и сможете создавать программы любой сложности.

УДК 004.43
ББК 32.973.26-018.1

Производственно-практическое издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Грег Перри, Дин Миллер

Программирование на C для начинающих
3-е издание

Директор редакции *Е. Капьёв*
Ответственный редактор *В. Обручев.* Художественный редактор *Е. Мишина*

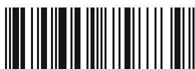
ООО «Издательство «Эксмо»
123308, Москва, ул. Зорге, д. 1. Тел. 8 (495) 411-68-86, 8 (495) 956-39-21.
Home page: **www.eksmo.ru** E-mail: **info@eksmo.ru**

Өндіруші: «ЭКСМО» АҚБ Баспасы, 123308, Мәскеу, Ресей, Зорге көшесі, 1 үй.
Тел. 8 (495) 411-68-86, 8 (495) 956-39-21
Home page: www.eksmo.ru E-mail: info@eksmo.ru.
Тауар белгісі: «Эксмо»
Қазақстан Республикасында дистрибьютор және өнім бойынша
арыз-талаптарды қабылдаушының
өкілі «РДЦ-Алматы» ЖШС, Алматы қ., Домбровский көш., 3«а», литер Б, офис 1.
Тел.: 8 (727) 2 51 59 89,90,91,92, факс: 8 (727) 251 58 12 вн. 107; E-mail: RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген.
Сертификация туралы ақпарат сайтта: www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ о техническом регулировании можно получить по адресу: http://eksmo.ru/certification/

Өндірген мемлекет: Ресей. Сертификация қарастырылмаған

Подписано в печать 13.01.2015. Формат 70x100 1/16.
Печать офсетная. Усл. печ. л. 29,81. Тираж экз. Заказ



ISBN 978-5-699-75873-9

© Райтман М.А., перевод на русский язык, 2015
© Оформление. ООО «Издательство «Эксмо», 2015

ОГЛАВЛЕНИЕ

Введение ... 10
Для кого предназначена эта книга? ... 10
Что отличает эту книгу от остальных? ... 11
Элементы дизайна этой книги ... 12
Как я могу развлечься, программируя на С? ... 13
Что я должен сделать сейчас? ... 13
Глава 1. Что такое программирование на С и почему это важно для меня? ... 14
Что такое программа? ... 14
Что вам понадобится для написания программ на языке С ... 16
Процесс программирования ... 20
Использование С ... 22
Абсолютный минимум ... 22
Глава 2. Ваша первая программа на С ... 24
Бесцеремонно убогий кусок кода ... 24
Функция `main()` ... 27
Виды данных ... 29
Символы в С ... 30
Числа в С ... 32
Подведем итог, рассмотрев еще один пример ... 34
Абсолютный минимум ... 35
Глава 3. Что делает эта команда? Пояснение кода с помощью комментариев ... 37
Комментирование программного кода ... 37
Вставка комментариев ... 39
Отступы ... 42
Еще один стиль комментариев ... 43
Абсолютный минимум ... 44
Глава 4. Мировая премьера: выход вашей программы на экраны ... 45
Использование функции printf() ... 45
Формат функции printf() ... 45
Печать строк ... 47
Управляющие последовательности ... 48
Символы преобразования ... 51
Обобщение и пример программного кода ... 53
Абсолютный минимум ... 54
Глава 5. Добавление переменных в программу ... 56
Виды переменных ... 56
Именование переменных ... 58
Объявление переменных ... 59
Сохранение данных в переменных ... 60
Абсолютный минимум ... 64
Глава 6. Добавление слов в программу ... 65
Символ конца строки ... 65
Длина строки ... 67

Символьные массивы: перечисление символов 68
Инициализация строк 71
Абсолютный минимум 73
Глава 7. #include и #define: увеличение мощности вашей программы 75
Включение файлов в программу 75
Использование директивы *#include* 78
Определение констант 79
Построение заголовочных файлов и программ 81
Абсолютный минимум 84
Глава 8. Взаимодействие с пользователем 85
Обзор функции *scanf()* 85
Запрос ввода данных функции *scanf()* 86
Проблемы с функцией *scanf()* 89
Абсолютный минимум 92
Глава 9. Числа: математические операции в С 94
Базовые арифметические операции 94
Порядок выполнения математических операций 98
Нарушение правил с помощью скобок 101
Повсеместное использование присваивания 101
Абсолютный минимум 104
Глава 10. Придание силы переменным с помощью присваиваний и выражений 105
Составной оператор присваивания 105
Следите за порядком! 110
Приведение типов переменных: Голливуд мог бы поучиться у С спецэффектам 111
Абсолютный минимум 112
Глава 11. Развилка на дороге: тестирование данных для выбора правильного пути 114
Тестирование данных 114
Использование оператора ветвления if 116
Иначе...: Использование *else* 119
Абсолютный минимум 124
Глава 12. Жонглирование опциями с помощью логических операторов 125
Становимся логичными 126
Избегаем негатива 131
Порядок логических операторов 134
Абсолютный минимум 136
Глава 13. Еще мешочек трюков: новые операторы для ваших программ 138
До свидания, *if ... else*, здравствуй, условный оператор 138
++ и *--*: операторы небольших изменений 142
Примеряем ситуацию 146
Абсолютный минимум 147
Глава 14. Повторение кода: использование циклов для экономии времени и сил 148
ПОКА мы повторяем 149
Использование цикла *while* 150
Использование цикла *do...while* 152
Абсолютный минимум 155

Глава 15. Еще один способ создания циклов ... 156
ДЛЯ и во имя повторения! ... 157
Работа с циклом for ... 159
Абсолютный минимум ... 165
Глава 16. Вход и выход из цикличного кода ... 167
Делаем перерыв на кофе-брейк ... 167
Давайте продолжим работать ... 170
Абсолютный минимум ... 173
Глава 17. Кейс для переключателя ... 174
Поворачиваем переключатель ... 175
break и switch ... 179
Размышления об эффективности ... 179
Абсолютный минимум ... 188
Глава 18. Усовершенствование ввода и вывода ваших программ ... 189
Функции *putchar()* и *getchar()* ... 189
Размышления о новых строках ... 193
Чуть быстрее: функция *getch()* ... 195
Абсолютный минимум ... 196
Глава 19. Получаем большее от строк ... 197
Функции, проверяющие символы ... 197
Верен ли регистр? ... 198
Функции, изменяющие регистр ... 202
Строковые функции ... 203
Абсолютный минимум ... 206
Глава 20. Высшая математика (для компьютера, не для вас!) ... 207
Практикум по математике ... 208
Еще несколько преобразований ... 209
Погружаемся в тригонометрию и другие сложные темы ... 210
Становимся непредсказуемыми ... 213
Абсолютный минимум ... 218
Глава 21. Работа с массивами ... 219
Повторение массивов ... 219
Запись значений в массивы ... 223
Абсолютный минимум ... 226
Глава 22. Поиск в массивах ... 227
Заполнение массивов ... 227
Находчики, хранители ... 228
Абсолютный минимум ... 235
Глава 23. Сортировка по алфавиту и упорядочение данных ... 236
Приберемся в доме: сортировка ... 236
Ускоренный поиск ... 242
Абсолютный минимум ... 247
Глава 24. Разгадка тайны указателей ... 249
Адреса памяти ... 249
Объявление переменных-указателей ... 250
Использование оператора разыменования * ... 252
Абсолютный минимум ... 256
Глава 25. Массивы и указатели ... 258
Названия массивов и указатели ... 258
Переход вниз по списку ... 260

Символы и указатели ... 261
Будьте внимательны с длиной ... 262
Массивы указателей ... 265
Абсолютный минимум ... 270
Глава 26. Максимизация памяти вашего компьютера ... 271
Размышления о динамической памяти ... 272
Но зачем мне нужна динамическая память? ... 273
Как я могу выделить динамическую память? ... 275
Если недостаточно динамической памяти ... 279
Освобождение динамической памяти ... 280
Множественное выделение памяти ... 281
Абсолютный минимум ... 285
Глава 27. Упорядочение данных с помощью структур ... 287
Объявление структуры ... 288
Запись данных в структурные переменные ... 293
Абсолютный минимум ... 297
Глава 28. Сохранение последовательных файлов на компьютере ... 299
Файлы на диске ... 299
Открытие файла ... 300
Использование файлов последовательного доступа ... 303
Абсолютный минимум ... 309
Глава 29. Сохранение файлов произвольного доступа на компьютере ... 310
Открытие файлов произвольного доступа ... 311
Перемещение по файлу ... 312
Абсолютный минимум ... 318
Глава 30. Организация программ с помощью функций ... 319
С функциями языка С приходит форма ... 319
Локальная или глобальная? ... 324
Абсолютный минимум ... 327
Глава 31. Передача переменных в функции ... 329
Передача аргументов ... 329
Методы передачи аргументов ... 330
Передача по значению ... 331
Передача по адресу ... 333
Абсолютный минимум ... 338
Глава 32. Возврат данных из функций ... 339
Возврат значений ... 339
Тип данных `return` ... 342
Последний шаг: прототип ... 343
Подведем итоги ... 346
Абсолютный минимум ... 346
Приложение А. Таблица ASCII ... 348
Приложение Б. Программа «Покер с обменом» ... 353
Об авторах ... 364
Благодарности ... 364
Предметный указатель ... 366

Посвящается Фрэн Хэттон, моей жене и лучшему другу,
всегда поддерживавшей мои мечты и являвшейся несокрушимой
опорной скалой в самый сложный год моей профессиональной карьеры.

ВВЕДЕНИЕ

Во введении

- Для кого предназначена эта книга?
- Что отличает эту книгу от остальных?
- Элементы дизайна этой книги
- Как я могу развлечься, программируя на С?
- Что я должен сделать сейчас?

Вам надоело видеть, как ваши друзья находят работу программистами С, в то время как вы остаетесь не у дел? Вы хотите изучить С, но для этого вам не хватает энергии? Нуждается ли ваш старый изношенный компьютер в горячем языке программирования, который мог бы взбудоражить и размять его чипы и платы? В таком случае данная книга — это то, что доктор прописал!

Книга «*Программирование на С. Руководство для новичков*» отличается от всех прочих книг о компьютерах тем, что она разговаривает с вами на вашем уровне, а не снисходит до вас со своих высот. Воспринимайте эту книгу как вашего лучшего друга, сидящего рядом и обучающего вас программированию на С. В книге «*Программирование на С. Руководство для новичков*» авторы пытаются объяснять, при этом не задавливая читателя интеллектом. Книга говорит с вами на простом и понятном русском, а не «компьютерском» языке. Короткие главы, наглядные схемы и время от времени юмористические высказывания проведут вас по лабиринтам языка программирования С быстрее, дружелюбнее и проще, чем любая другая книга на полке магазина.

Для кого предназначена эта книга?

Эта книга для начинающих. Если вы никогда не занимались программированием, то она для вас. Предполагается, что у вас отсутствует всяческие знания по концепциям программирования. Даже если вы не можете правильно написать слово С, используя эту книгу, вы все равно сможете научиться программировать на языке С.

Слово *новичок* в разное время может иметь разные значения. Возможно, вы уже пытались изучить язык C, но тщетно — и вы прекратили попытки. Очень большое количество книг и курсов по программированию преподносят C гораздо более техническим языком, чем он есть на самом деле. Возможно, вы также программировали на других языках, но являетесь новичком в программировании на C. Если это так, то продолжайте читать, так как через 32 короткие главы верующий да познает язык программирования C.

Что отличает эту книгу от остальных?

Эта книга не создает облако из внутренних технических нюансов языка C, не нужных начинающему программисту. Мы, наоборот, твердо уверены в том, что вводным принципам необходимо обучать дотошно и неспешно. После того, как вы в полной мере овладеете основными принципами, «более тяжелые» аспекты не покажутся вам такими тяжелыми.

Язык C может показаться очень непонятным, даже криптографическим, и крайне сложным. Многие люди предпринимают несколько попыток овладения этим языком программирования. Проблема заключается в следующем: любой предмет, будь то операция на головном мозге, сортировка почта или программирование на C, будут казаться простыми только в том случае, если вам их правильно объяснили. На самом деле никто не сможет вас ничему научить, так как вы должны научить *самих себя*. Однако, если преподаватель, книга или видеоуроки не делают предмет простым и веселым, вы вряд ли захотите изучать этот предмет.

Мы вызываем вас на поединок, который будет заключаться в том, что вы найдете более прямолинейный подход к языку C, чем тот, что мы предлагаем в книге «*Программирование на C. Руководство для новичков*». Если вы сможете это сделать — позвоните любому из авторов этой книги: мы будем рады прочитать вашу методику. (А вы думали, мы предложим вернуть вам деньги?) Если говорить серьезно, то мы попытались предоставить вам вспомогательные материалы различного вида, те материалы, которые вы можете найти в самых разнообразных источниках.

Однако, самое большое преимущество этой книги заключается в том, что мы на самом деле *любим* писать программы на языке C, но больше этого мы любим обучать программированию на C. Мы надеемся, что вы тоже полюбите язык C.

Элементы дизайна этой книги

Как и многие другие книги, посвященные компьютерам, эта книга содержит большое количество полезных советов, подсказок, предупреждений и прочего. При чтении книги вы повстречаетесь с большим количеством заметок и выносок на полях, которые призваны привлечь ваше внимание к какой-либо информации.

СОВЕТ

Большое количество особых приемов и советов, помещенных в эту книгу (а их на самом деле много), выделены примечанием со словом «Совет». Когда какая-либо действительно тонкая особенность или прием написания кода совпадает с тематикой того, о чем вы читаете, слово «Совет» указывает, что вы можете получить определенное преимущество.

ПРИМЕЧАНИЕ

В языке C есть определенные аспекты, требующие более глубокого понимания, чем прочие темы. Примечание расскажет вам нечто такое, о чем вы, возможно, даже не могли подумать, например альтернативный вариант использования обсуждаемой темы.

ВНИМАНИЕ

Врезка «Внимание» указывает на потенциальную возможность возникновения проблем, с которыми вы можете столкнуться при изучении какой-то конкретной обсуждаемой темы. Примечание такого рода выдает предупреждение, на которое вам следует обратить внимание, или же предоставляет вариант исправления проблем, которые могут возникнуть.

Каждая глава заканчивается обзором ключевых моментов, которые вам необходимо запомнить. Одна из ключевых особенностей, связывающих все главы воедино — это раздел, озаглавленный «Абсолютный минимум». Такое резюме текста главы устанавливает первоочередную цель главы, приводит пример программного кода, подчеркивающего изученные вами концепции, а также содержит анализ, объясняющий работу приведенного примера программного кода. Надеемся, вы найдете такие резюме текста главы, которые помещаются нами в книге, начиная с главы 2, полезными обобщениями основных моментов, обсуждаемых в той или иной главе.

В книге используются следующие типографические соглашения.

- Строки кода, переменные и любой текст, который вы увидите на экране монитора, напечатан с использованием `моноширинного` шрифта.
- Заполнители строк форматирования приводятся *`моноширинным курсивом`*.
- Части программного вывода, введенные пользователем, выделяются **`жирным моноширинным`** шрифтом.
- Новые термины выделяются *курсивом*.
- Необязательные параметры и пояснения синтаксиса заключены в квадратные ([ ]) скобки. При использовании этих необязательных параметров включать квадратные скобки в программный код не нужно.

Как я могу развлечься, программируя на C?

В приложении Б содержится полный и работающий исходный код программы «Покер с обменом». Мы попытались максимально сократить текст этой программы, не принося при этом в жертву читаемость кода и функциональность игрового сюжета. Кроме того, эта игра должна оставаться достаточно обобщенной с тем, чтобы ее можно было использовать с различными компиляторами языка C. Потому в этой игре вы не найдете привлекательной графики, однако, изучив язык C с целью усовершенствования этой программы, вы с легкостью сможете получить доступ к специфическим графическим и звуковым возможностям, а также процедурам ввода данных своего компьютера.

В данной программе используется максимальное количество содержимого этой книги. Практически каждая тема, обсуждаемая в книге, находит свое применение в игре «Покер с обменом». Слишком большое количество книг не предлагает ничего, кроме образцов кода. Игра «Покер с обменом» дает вам возможность увидеть «общую картину». По мере чтения этой книги вы начнете понимать все большее и большее количество кода данной игры.

Что я должен сделать сейчас?

Перевернуть страницу и начать изучать язык C.

Глава 1
ЧТО ТАКОЕ ПРОГРАММИРОВАНИЕ НА С И ПОЧЕМУ ЭТО ВАЖНО ДЛЯ МЕНЯ?

В этой главе

- Понимание основ программирования на С
- Поиск и установка компилятора языка С
- Изучение шагов процесса программирования

Несмотря на то, что многие люди считают язык программирования С сложным для изучения и использования, вы вскоре убедитесь, что они не правы. О языке С говорят как о языке повышенной, почти криптографической сложности. В общем, он может быть сложным, однако хорошо написанная программа на языке С так же удобочитаема, как и программа на любом другом языке. Сегодня требования к программистам и разработчикам очень высоки, а изучение языка С может стать эффективной базой для получения навыков, необходимых в самых разнообразных отраслях, в том числе создании приложений, программировании игр и так далее.

Если вы никогда в жизни еще не писали компьютерных программ, то эта глава будет для вас отличным началом, так как в ней вы сможете изучить вводные концепции программирования, получить объяснение того, что такое программа, а также прочесть краткую историю языка программирования С. Приготовьтесь восхищаться! С — это язык программирования с богатыми возможностями.

Что такое программа?

Компьютер не умен. Верите вы нам или нет, но даже в самые тяжелые дни вы на световые годы разумнее своего компьютера. Вы можете думать и можете сообщать компьютеру, что он должен сделать. С одним

компьютер справляется блистательно: он будет подчиняться всем вашим приказам. Ваш компьютер будет сидеть за столом днями и ночами напролет, обрабатывая передаваемые ему данные, причем это ему не наскучит и он не попросит почасовой оплаты.

Компьютер не может самостоятельно решать, что ему делать. Компьютеры не умеют думать самостоятельно, поэтому *программисты* (люди, говорящие компьютерам, что они должны делать) должны давать компьютерам максимально детализированные инструкции. Без инструкций компьютер бесполезен, при получении неверных инструкций, он не сможет успешно выполнить переданное ему задание. Без детальных инструкций компьютер не сможет обработать вашу зарплатную ведомость, равно как и автомобиль не сможет автоматически и самостоятельно запустить двигатель и проехаться по кварталу. Набор детализированных выражений, передаваемых вами компьютеру, когда вы желаете, чтобы компьютер выполнил какое-либо задание, называется *программой*.

ПРИМЕЧАНИЕ

Текстовые процессоры, приложения, электронные таблицы и компьютерные игры не являются ничем бо́льшим, чем компьютерная программа. Без этих программ компьютер просто сидел бы на вашем столе, не зная, что следует предпринять далее. Программа — текстовый редактор содержит список детальных инструкций, написанных на компьютерном языке, таком как C, говорящих вашему компьютеру, как быть редактором текста. Когда вы программируете, вы говорите компьютеру следовать инструкциям, содержащимся в вашей программе.

Для своего компьютера, планшета или телефона вы можете приобрести или скачать тысячи различных программ, но когда бизнесу требуется компьютер, который смог бы выполнять какую-то специфическую задачу, бизнес нанимает программистов и разработчиков для создания программного обеспечения, которое соответствовало бы спецификациям, отвечающим потребностям бизнеса. Вы можете заставить свой компьютер или мобильное устройство делать очень многое, но вы можете и не найти программу, которая бы выполняла именно то, что нужно вам. Данная книга призвана спасти вас от этой дилеммы. После того, как вы изучите язык C, вы сможете писать программы, говорящие компьютеру, как себя вести.

СОВЕТ

Компьютерная программа сообщает вашему компьютеру, как сделать то, что хотите вы. Как шеф-повар для приготовления блюда нуждается в рецепте, компьютер нуждается в инструкциях для произведения какого-то результата. Рецепт — это набор детальных инструкций, которые, при условии, что они правильно написаны, описывают правильную последовательность и содержание шагов по приготовлению какого-то конкретного блюда. Это как раз то, чем является компьютерная программа для вашего компьютера.

При *запуске* или *выполнении* программы генерируют определенный *вывод*.

Приготовленное блюдо — это вывод из рецепта, а текстовый редактор или приложение — это вывод, продуцируемый запущенной программой.

ВНИМАНИЕ

Аналогично тому, когда повар использует неверный ингредиент или пропускает шаг из рецепта и в результате получается несъедобное блюдо, ваша программа не будет работать, если вы допустите опечатку в коде или пропустите шаг.

Что вам понадобится для написания программ на языке С

Прежде чем вы сможете написать и запустить свою программу на С на компьютере, вам потребуется *компилятор* языка С. Компилятор С принимает написанную вами программу на языке С и занимается ее *построением* или *компиляцией* (технические термины, обозначающие процесс превращения вашей программы в программу, понятную компьютеру), что позволяет вам запустить скомпилированную программу, когда вы будете готовы посмотреть на результаты ее выполнения. К счастью, существует большое количество бесплатных пакетов программного обеспечения, позволяющих вам редактировать и компилировать программы, написанные на языке С. Простой поисковый запрос выдаст вам список результатов с такими программами. При написании этой книги мы использовали компилятор Code::Blocks (**http://www.codeblocks.org**).

СОВЕТ

Если вы выполните поиск по запросу «Компиляторы С», то вы увидите большое количество бесплатных опций, в том числе предложения от компаний Borland и Майкрософт. Тогда почему же при написании книги мы пользовались компилятором Code::Blocks? Потому что для этого продукта доступны версии для Windows, Mac и Linux, поэтому вы сможете использовать подходящую версию программного обеспечения вне зависимости от установленной на вашем компьютере операционной системы. Однако вы можете выбрать любую другую программную среду, которая покажется вам лучше.

Перейдя на страницу Code::Blocks в интернете и прочитав самое первое предложение, вы можете слегка (или не на шутку) заволноваться:

```
The open source, cross platform, free C++ IDE*.
```

Термин *Open source* (открытый, с открытым исходным кодом) применяется для обозначения такого программного обеспечения, исходный код которого может быть изменен или улучшен пользователем. (В ближайшее время вы не будете заниматься ничем подобным, поэтому не берите в голову.) *Cross-platform* (кросс-платформенный) — это прилагательное, означающее, что данное программное обеспечение может запускаться в различных операционных системах. Однако новичков должно заботить только то, чтобы программное обеспечение запустилось на вашей платформе. Термин *Free* (бесплатный), как нам кажется, понятен и без дополнительных объяснений, а *IDE* — это аббревиатура для английского словосочетания «интегрированная среда разработки» (Integrated Development Environment), которое, в свою очередь означает, что вы можете писать, редактировать и отлаживать свои программы без необходимости переключения между программным обеспечением. Скоро мы вернемся к отладке.

Не стоит паниковать по поводу части предложения С++. С помощью программной среды Code::Blocks вы сможете писать программы как на языке С, так и на С++ («С плюс-плюс»). Сегодня найти компилятор С значительно сложнее: в большинстве случаев компиляторы оснащены поддержкой более совершенной версией языка, называемой С++. Поэтому при поиске компилятора языка С вы почти всегда будете находить комбинацию компиляторов С и С++, причем возможность работы с язы-

* Открытая кросс-платформенная бесплатная интегрированная среда разработки С++.

ком C++ зачастую дополнительно подчеркивается. Хорошая новость в том, что, изучив язык C, вы уже будете иметь компилятор C++, и вам не придется заново изучать все нюансы и подробности новой IDE.

На рис. 1.1 изображена главная страница сайта Code::Blocks. Для загрузки интегрированной среды разработки C/C++ перейдите по ссылке **Downloads** (Загрузки) в разделе **Main** (Главное) в колонке слева.

Рис. 1.1. Главная страница Code::Blocks. Вам необходимо обратить внимание на ссылку **Downloads** (Загрузки)

После перехода по ссылке **Downloads** (Загрузки) вы попадете на страницу, более подробно описывающую опции для загрузки, их три: **Binaries** (Бинарные файлы), **Source** (Исходный код) и **SVN**. Последние две опции предназначены для опытных программистов, поэтому щелкните мышью по опции **Binaries** (Бинарные файлы).

ПРИМЕЧАНИЕ

При установке программы обратите внимание на два момента. Первый: снимки экрана, помещенные в книге, возможно, будут незначительно отличаться от того, что вы увидите в интернете, так как Code::Blocks постоянно работает над улучшением своего программного обеспечения, поэтому числа (обозначающие номера версий) постоянно увеличиваются. Версия программного обеспечения Code::Blocks, использованная при написании книги — 10.05, однако при последней сверке номер новейшей версии был 12.11, возможно, когда вы будете читать это примечание, номер текущей версии опять изменится. Второе: если вы пользователь ОС Windows, убедитесь в том, что загружаете самый большой из доступных файлов (содержащий в своем имени буквосочетание «mingw»). В этой версии есть инструменты для отладки, которые пригодятся вам, когда вы станете про-СИ-женным программистом. (Поняли шутку? Нет? Я что, один такой?)

На следующей странице представлены различные опции, зависящие от вашей операционной системы. Если вы выбрали загрузку для ОС Windows, выберите вторую опцию для загрузки, как показано на рис. 1.2. Иметь полную версию компилятора и отладчика будет вам полезно.

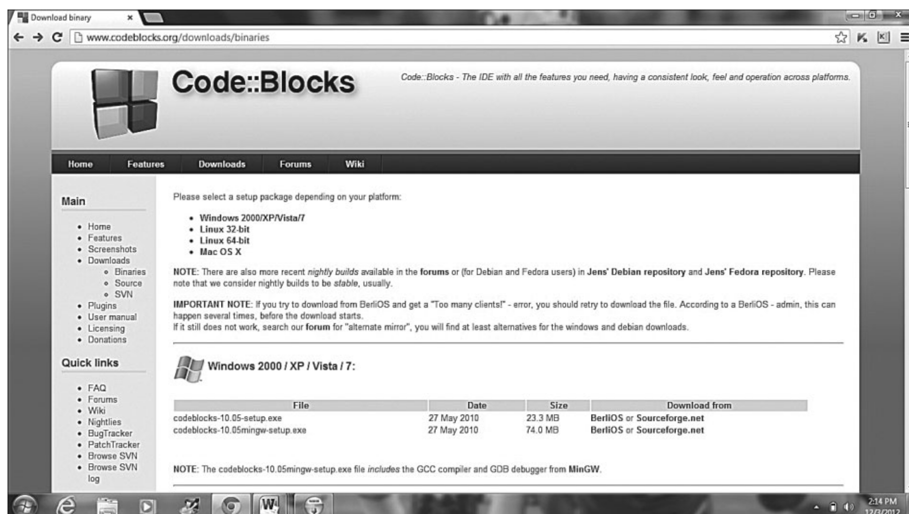


Рис. 1.2. Выбор загрузки IDE для Windows. Вы также можете выбрать загрузку исходного кода

Начав загрузку, можете смело отправиться на кухню выпить чайку, так как размер программного файла велик и для его загрузки может потребоваться несколько минут. По завершении загрузки запустите файл и согласитесь со всеми опциями по умолчанию. Только опытным программистам может потребоваться изменять эти настройки. Через некоторое время программа Code::Blocks будет запущена на вашем компьютере. После того, как вы выйдете из окна **Tip of the Day** (Совет дня) и установите ассоциацию с этой программой для всех файлов .C и .CPP, вы можете закрыть окно настройки сценариев. На экране у вас должно остаться приветственное окно программного обеспечения, как показано на рис. 1.3.

ПРИМЕЧАНИЕ

Создаваемая вами программа на языке С называется *исходным кодом*. Компилятор переводит исходный код, написанный на языке С, на *машинный язык*. Компьютеры состоят не из чего иного, как тысяч электрических переключателей, которые могут быть либо *вклю-*

чены, либо *выключены*. Поэтому компьютерам необходимо давать инструкции, написанные только в *бинарном* числовом формате. Латинская приставка «би-» переводится на русский язык как «два», при этом два состояния электрической схемы называются *бинарными состояниями*. Гораздо проще использовать компилятор языка С для перевода ваших программ с языка С в последовательность нулей и единиц, каждая из которых представляет указание состояния внутреннего переключателя «выключено» или «включено» соответственно, чем писать такую последовательность самостоятельно.

Рис. 1.3. Добро пожаловать в ваш программистский дом!

Процесс программирования

При написании программ большинство людей придерживаются следующего алгоритма действий.

1. Решить, что конкретно должна выполнять ваша программа.
2. Воспользоваться *редактором* для написания и сохранения программных инструкций. Редактор очень похож на текстовый процессор, позволяющий вам создавать и редактировать тексты (хотя, конечно, не обладает столь привлекательным интерфейсом). Все популярные компиляторы языка С, кроме собственно компилятора, также включают в себя и интегрированный редактор кода. Имена всех программных файлов, написанных на языке С, имеют расширение .С.

3. Скомпилировать программу.
4. Выполнить проверку ошибок в программе. Если таковы найдены, исправить их и вернуться к шагу 3.
5. Запустить программу.

ПРИМЕЧАНИЕ

Ошибки в компьютерной программе многие называют *багами* (от английского слова «*bug*» — жук. — *Примеч. пер.*). Процесс избавления от ошибок в программе называется *отладкой* программы (что на английском языке звучит как «debugging», дословно «дезинсекция». — *Примеч. пер.*).

Уделите какое-то время исследованию программы Code::Blocks или любого другого компилятора, который вы решили установить на своем компьютере. Тщательно проработанная интегрированная среда разработки позволяет вам с легкостью совершать все пять вышеописанных шагов, при этом используя для всех этих задач единую среду. Вы можете компилировать свою программу, просматривать сообщения о выявленных ошибках, исправлять их, запускать программу и просматривать результаты ее выполнения. Все это вы можете делать из единого окна, используя стандартизированный набор пунктов меню.

ВНИМАНИЕ

Если вы никогда прежде не занимались программированием, все вышеописанное может показаться вам запутанным и сложным. Не волнуйтесь. Большинство современных компиляторов С поставляются с удобным в использовании учебным руководством, которое вы сможете использовать для того, чтобы изучить основные аспекты использования редактора программного кода и команд компилятора.

На случай, если вы все еще не поняли, в чем состоит необходимость использования компилятора, поясним, что исходный код подобен сырьевым материалам, необходимым вашему компьютеру. Компилятор подобен машине или станку, обрабатывающему сырье и преобразующему его в конечный продукт, компилированную программу, которую ваш компьютер сможет понять.

Использование C

C гораздо более эффективен, чем большинство языков программирования. Кроме того, это достаточно небольшой язык программирования. Иными словами, при изучении C вам не придется разучивать большое количество *команд*.

При чтении этой книги вы изучите команды и прочие элементы языка C, такие как операторы, функции и директивы препроцессора.

По причине существования большого количество версий языка C комитет ANSI (American National Standards Institute — Американский национальный институт стандартизации) создал набор стандартов (называемых ANSI C), обязательных для всех версий языка C.

Если вы запускаете собственные программы, используя компилятор языка ANSI C, можете быть уверены, что ваши программы, написанные на C, могут быть скомпилированы на любом компьютере, оборудованном компилятором языка ANSI C.

СОВЕТ

В 1983 году ANSI создал комитет X8J11 для разработки стандартной версии языка C, который в дальнейшем стал известен под именем ANSI C. Самая последняя версия языка ANSI C C11 была формально принята в 2011 году.

После компиляции программы на языке C с помощью компилятора вы сможете запускать скомпилированную программу на любом компьютере, совместимом с вашим, вне зависимости от того, оснащен ли этот компьютер компилятором языка ANSI C. «Круто! — возможно, скажете вы. — Но когда я, наконец, начну писать мою первую программу на C, когда смогу скомпилировать и запустить ее?» Не беспокойтесь, глава 2 сопроводит Вас в вашем первом путешествии в мир программирования на C.

Абсолютный минимум

В данной главе мы вкратце представили вам язык программирования C и помогли вам выбрать компилятор для редактирования, отладки и запуска ваших программ.

Ниже перечислены несколько ключевых моментов, которые необходимо запомнить.

Загрузите и установите компилятор С на свой компьютер.

Приготовьтесь к изучению языка программирования С.

Не волнуйтесь, что язык С может показаться вам слишком сложным. В этой книге мы постарались разбить концепции языка С на удобоваримые небольшие сегменты. Немножко попрактиковавшись, вы с легкостью их освоите.

Глава 2
ВАША ПЕРВАЯ ПРОГРАММА НА C

В этой главе

- Написание вашей первой программы
- Использование функции `main()`
- Определение типов данных

В этой главе вы повстречаетесь с вашей первой программой на языке C! Пожалуйста, не пытайтесь понять каждый символ программ на C, обсуждаемых в этой главе. Расслабьтесь и просто ознакомьтесь с тем, как выглядит и ощущается язык программирования C. Через некоторое время вы научитесь узнавать общие элементы, характерные для всех программ на C.

Бесцеремонно убогий кусок кода

В этой секции мы продемонстрируем вам короткую, но завершенную программу на языке C, а также обсудим другую программу, которую вы встретите в приложении Б. В обеих программах вы найдете как схожие, так и отличающиеся элементы. Наша первая программа будет чрезвычайно простой:

```
/*Выводит сообщение на экране*/
#include <stdio.h>
main()
{
  printf("Just one small step for coders. One giant
leap for");
  printf(" programmers!\n");
  return 0;
}
```

Теперь откройте установленное на вашем компьютере программное обеспечение для программирования и введите с клавиатуры программу точно так, как приводится в листинге сверху. Все просто, не правда ли? Да, возможно, но не в случае, если вы используете новый для себя компилятор. При первом запуске программы Code::Blocks вы будете встречены приветственным сообщением в окне **Tip of the Day** (Совет дня). Эти советы окажутся вам полезными немного позднее, но сейчас вы можете убрать это окно, просто щелкнув мышью по кнопке **Close** (Закрыть).

Для создания собственной программы щелкните левой кнопкой мыши по пункту меню **File** (Файл), наведите указатель мыши на пункт **New** (Новый) раскрывшегося меню и выберите пункт **Empty File** (Пустой файл) из числа опций открывшегося подменю. Теперь у вас есть приятный чистый файл, в котором вы можете начать писать вашу программу о семи строках.

После написания вам потребуется скомпилировать или построить вашу программу. Чтобы сделать это, щелкните мышью по маленькой желтой иконке с изображением шестеренки в левом верхнем углу. Если вы ввели программу точно как показано в тексте и не сделали ошибок, то теперь вы можете запустить программу, щелкнув мышью по маленькой зеленой стрелке, направленной вправо, расположенной рядом с изображением шестеренки. (Следующая кнопка, на которой изображена и стрелка и шестеренка, автоматически запускает программу после компиляции, что несколько упрощает вашу жизнь, уменьшая неимоверное количество щелчков мышью с двух до одного.)

Скомпилировав (или построив) программу и запустив ее, на вашем экране вы увидите нечто похожее на то, что изображено на рис. 2.1.

ПРИМЕЧАНИЕ

Вывод этой строки сообщения потребовал много работы! На самом деле из семи строк программы только две строчки (те, что начинаются с `printf`) производят действия, генерирующие вывод сообщения на экран. Все остальные строки делают лишь так называемую работу по дому и встречаются почти во всех программах, написанных на языке C.

Чтобы лицезреть гораздо более длинную программу, пробегитесь взглядом по приложению В. Несмотря на то, что покерная программа,

приводимая в Приложении, занимает несколько страниц, она все же содержит элементы, похожие на те, что есть в коде только что продемонстрированной программы.

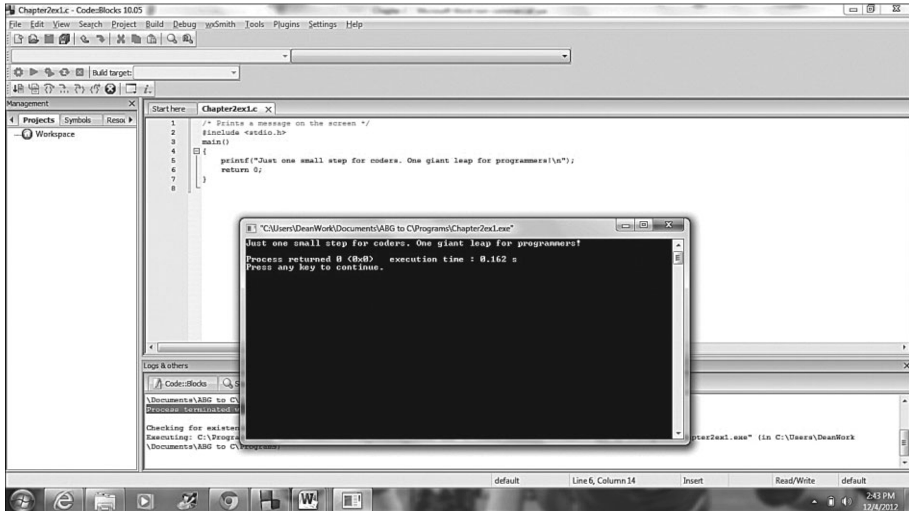


Рис. 2.1. Вывод вашей первой программы

Просмотрите текст обеих обсуждаемых программ и попытайтесь найти похожие элементы. Первое, что, возможно, бросилось вам в глаза — это использование фигурных скобок ({ }), квадратных скобок ([ ]) и обратных слешей (\). При вводе программ в компилятор языка C будьте внимательны, так как языку C явно не понравится, если вместо фигурной скобки вы напечатаете квадратную ([).

ВНИМАНИЕ

Кроме того, что вам необходимо следить за тем, не напечатали ли вы неподходящий символ, вам нужно также быть предельно внимательным при наборе кода программы в текстовом процессоре с последующей вставкой в среду разработки IDE. При написании этой книги предыдущую программу мы сначала набрали в текстовом редакторе Word, затем скопировали ее и вставили в программу Code::Blocks. При компиляции программы система выдала нам сразу несколько сообщений об ошибке, так как кавычки, используемые нами в коде программы, были преобразованы программой Word в так называемые умные кавычки (позволяющие визуально отличить начало и конец цитаты) и компилятор, соответственно, не смог их распознать. После того как я удалил кавычки в окне редактора кода Code::Blocks

и заново напечатал их в окне этого же редактора, код был скомпилирован без возникновения каких-либо ошибок. Поэтому, если при компиляции программ система выдает вам сообщения об ошибках, мы рекомендуем вам удостовериться в том, что причина этих ошибок не кроется в неправильных кавычках.

Однако язык С придирчив не ко всему. Так, например, большинство пробелов и промежутков между участками кода, которые вы можете встретить в программе на языке С, упрощают понимание текста программы для человека, но не для компилятора С. При написании программ оставляйте пустые строки между блоками программ и добавляйте отступы для удобочитаемости программы и облегчения поиска нужного участка кода.

СОВЕТ

Для создания краевых отступов воспользуйтесь клавишей табуляции (**Tab**) на клавиатуре, не стоит вместо этого пользоваться клавишей **Пробел**. Большинство редакторов С позволяют настроить ширину отступов, то есть количество пробелов, появляющихся в тексте при нажатии клавиши табуляции. Некоторые строки программ на языке С могут быть очень длинными, поэтому установка ширины краевого отступа равным трем пробелам позволит вам оставлять достаточный отступ, не увеличивая чрезмерно длину строки.

Согласно требованиям языка С, при написании всех команд и имен предопределенных (встроенных) функций необходимо использовать символы нижнего регистра (что такое функция, вы прочтете в следующем разделе). Можно сказать, что символы верхнего регистра используются в основном только в строках, содержащих ключевое слово `#define`, либо в сообщениях, которые вы хотите вывести на экран.

Функция *main()*

Самая важная часть программы, написанной на языке С, — это функция `main()`*. В обеих обсуждаемых ранее программах присутствует функция `main()`. Несмотря на то, что на данном этапе различение понятий «команда» и «функция» не принципиально, `main()` является

* Main (*англ.*) — основной, главный. — *Примеч. пер.*

функцией языка С, а не командой. Функция — это не что иное, как некая процедура, выполняющая какое-либо задание. Некоторые функции включены в комплект поставки языка С, а некоторые функции вы будете создавать самостоятельно. Программы на языке С могут состоять из нескольких функций. В каждой программе на С *всегда* должна присутствовать функция `main()`. От команды функцию отличают круглые скобки, следующие сразу после имени. Так выглядят функции:

```
main()    calcIt()    printf()    strlen()
```

а так команды:

```
return    while    int    if float
```

При чтении других книг, руководств или веб-страниц, посвященных программированию на С, вы можете встретить ситуацию, когда автор решил опустить скобки при описании функции. Например, вы можете читать статью, в которой название функции приводится как `printf`, а не `printf()`. В скором времени вы научитесь распознавать имена функций, поэтому подобное различие не будет для вас сколько-нибудь существенным. Однако чаще всего авторы публикаций стараются сделать различие между функциями и не функциями наиболее очевидным, поэтому обычно при чтении таких публикаций вы будете встречать круглые скобки.

ВНИМАНИЕ

В имени одной из вышеприведенных функций — `calcIt()` — содержится буква верхнего регистра, хотя в предыдущем разделе мы говорили, что использования заглавных букв следует по возможности избегать. Однако если имя функции состоит из нескольких частей, как, например, имя функции `doReportPrint()`, то разделение этих частей имени при помощи заглавных букв является общепринимой практикой, так как это позволяет повысить удобочитаемость кода. (Обратите внимание, что в именах функций недопустимо использовать пробелы.) Старайтесь избегать написания слов только заглавными буквами, но нечастое их употребление для повышения удобочитаемости кода вполне желательно.

Имя функции `main()`, а также имена практически всех встроенных функций С должны содержать только буквы нижнего регистра. При создании собственных функций вы можете использовать заглавные буквы, однако большинство программистов, работающих на языке С,

придерживаются соглашения об использовании букв нижнего регистра.

Аналогично тому, как главная страница сайта является для вас отправной точкой поиска информации на этом сайте, функция `main()` всегда является местом, откуда компьютер начинает выполнение написанной вами программы. Даже если функция `main()` — не первая перечисленная функция в вашей программе, данная функция все равно определяет начало процесса выполнения программы. Поэтому для повышения удобочитаемости всегда помещайте функцию `main()` первой. Программы, приводимые в нескольких следующих главах, содержат только функцию `main()`. По мере улучшения ваших навыков программирования на С вы начнете понимать, почему добавление собственных функций после функции `main()` увеличит ваши возможности как программиста. В главе 30 рассказывается о том, как писать ваши собственные функции.

После ключевого слова `main()` вы всегда встретите открывающуюся фигурную скобку (`{`). Далее, когда в тексте программы вы увидите соответствующую закрывающуюся фигурную скобку (`}`), это будет означать, что блок функции `main()` завершен. Внутри функции `main()` вы также можете встретить еще несколько пар открывающихся и закрывающихся фигурных скобок. Чтобы попрактиковаться в поиске, взгляните еще раз на длинную программу, помещенную нами в приложение В. В коде этой программы `main()` — первая функция, за которой также следует еще несколько дополнительных функций с фигурными скобками и программным кодом между ними.

ПРИМЕЧАНИЕ

Выражение `#include <stdio.h>` необходимо включать практически во все программы на С, так как оно помогает выводить на экран и получать данные. На данном этапе просто не забывайте помещать выражение `#include <stdio.h>` в код программы перед функцией `main()`. Понять всю важность выражения `#include` вы сможете, изучив главу 7.

Виды данных

Создаваемые вами программы на языке С будут использовать различные данные, представляющие собой числа, символы и слова. Про-

граммы обрабатывают данные и преобразуют их в значимую информацию. Несмотря на существование большого количества видов данных, наиболее популярных типов данных, используемых в программировании на языке С, три:

- Символы
- Целые числа
- Числа с плавающей точкой (их еще называют *вещественными числами*)

СОВЕТ

Возможно, вы воскликнете: «Как хорошо я должен буду выучить математику? Мы ведь так не договаривались!» Ну, вы можете расслабиться, так как язык С возьмет всю математику на себя, чтобы программировать на С вам даже не нужно будет уметь складывать 2+2. Однако вам понадобится разбираться в типах данных, чтобы вы смогли выбрать нужный тип для использования в своей программе.

Символы в С

Символом в С называется любой отдельный символ, отображаемый вашим компьютером. Вашему компьютеру известны 256 различных символов, каждый из которых вы можете найти в таблице ASCII (читается как «аск-и») в приложении А «Таблица ASCII». Все, что ваш компьютер может вывести на печать, может представлять собой символ. Так, все нижеследующее является примерами символов:

```
A a 4 % Q ! + = ]
```

ПРИМЕЧАНИЕ

Американский национальный институт стандартов (ANSI), разработавший язык ANSI С, также разработал код таблицы ASCII.

СОВЕТ

Даже пробел является символом, поэтому, аналогично тому, как программа на С следит за каждым вводимым или выводимым символом, буквой или цифрой, ей необходимо присматривать и за любым количеством пустого места, которое может ей понадобиться.

Таким образом, вы можете видеть, что любая буква, цифра или пробел воспринимается С как символ. Конечно, 4 выглядит как число и иногда им и является, но это в том числе и символ. Если вы обозначите цифру 4 как символ, то вы не сможете производить с ней никаких математических операций. Если другую цифру 4 вы обозначите числом, то с ней вы сможете производить математические вычисления. То же самое касается и специальных символов. Знак плюс (+) — это символ, но знак плюс также позволяет произвести операцию сложения (ну вот опять меня понесло, и я опять заговорил о математике).

Все символы С заключаются в *апострофы* ('), некоторые называют апострофы *одинарными кавычками*. Апострофы позволяют отличить тип данных символы от прочих типов данных, например чисел и знаков математических операций. Например, в программе на языке С все нижеследующее будет являться символами:

```
'A' 'a' '4' '%' ' ' '-'
```

Ничто из перечисленного ниже не может относиться к символьному типу данных по причине отсутствия кавычек:

```
A a 4 % -
```

СОВЕТ

Ничто из перечисленного ниже также не относится к символьному типу данных, так как только одиночные символы, но не их сочетания, можно помещать между одинарных кавычек:

```
'С - это весело '
'С - это сложно'
'Лучше бы я занялся яхтингом'
```

Первая программа из этой главы содержит символ '\n'. Поначалу вы могли подумать, что \n не является единым символом, однако это одно из немногих сочетаний символов, воспринимаемых языком С как единый символ. Вы поймете смысл вышесказанного чуть позже.

Если вам необходимо указать более одного символа (за исключением специальных, таких, как только что описанный нами символ \n), заключите символы в *двойные кавычки* ("). Группа, состоящая из нескольких символов называется *строкой*. Ниже приведен пример строки С:

```
"Изучать язык С весело".
```

ПРИМЕЧАНИЕ

На данный момент это все, что вам нужно знать про символы и строки. В главах с 4-й по 6-ю вы узнаете, как использовать их в программах. Узнав, каким образом можно сохранять символы в переменных, вы поймете, почему апострофы и кавычки столь важны.

Числа в С

Хотя, возможно, вы и не думали об этом раньше, но числа могут иметь разные размеры и принимать различные формы. В вашей программе на С должен быть предусмотрен способ хранения чисел, независимо от того, как выглядят эти числа. Для хранения чисел используются числовые переменные. Впрочем, прежде чем говорить о переменных, давайте вспомним виды чисел.

Числа без дробной части называются *целыми*. У целых чисел нет точки — дробного разделителя*. Любое число без дробного разделителя есть целое число. Все указанные ниже числа являются целыми:

```
10 54 0 -121 -68 752
```

ВНИМАНИЕ

Никогда не начинайте вводить целое число с 0, если вводимое число не есть 0, в противном случае компилятор языка С воспримет такое число как число шестнадцатеричной или восьмеричной системы счисления. Шестнадцатеричная и восьмеричная системы счисления — это странный способ представления чисел, так, число `053` относится к восьмеричной системе, а число `0x45` — к шестнадцатеричной системе счисления. Если вы не понимаете, что все это значит, то просто пока запомните, что компилятор языка С вас четырежды четвертует на *шестнадцать* кусков, если вы устроите беспорядок с нулями.

Числа с дробным разделителем называются *числами с плавающей точкой (запятой)*. Все указанные ниже числа относятся к категории чисел с плавающей точкой:

```
547.43 0.0 0.44384 9.1923 -168.470 22
```

---

* В большинстве европейских языков, в том числе английском, на основе которого был разработан язык программирования С, при написании десятичной дроби, например 3,14, дробную часть от целой принято отделять точкой, т.е. в языке С запись «три целых четырнадцать сотых» должна выглядеть следующим образом: 3.14.

СОВЕТ

Как видите, в случае с числами с плавающей точкой, начальные нули не вызывают каких-либо трудностей.

Выбор целочисленного или дробного типа данных зависит от данных, с которыми будет работать ваша программа. Для представления некоторых показателей, таких как возраст и количество, достаточно использовать целые числа, но для других показателей, например денежные суммы или вес, может потребоваться более высокая точность указания, которую могут предоставить вам числа с плавающей точкой. Внутренне системы хранения целых и дробных чисел в С устроены по-разному. Как показано на рис. 2.2, на хранение числа с плавающей точкой может потребоваться в два раза больше памяти, чем на хранение целочисленного значения. Поэтому, если вы можете обойтись использованием целых чисел, мы рекомендуем поступить именно так: используйте числа с плавающей точкой только для показателей, которым необходимо представление в виде десятичной дроби.

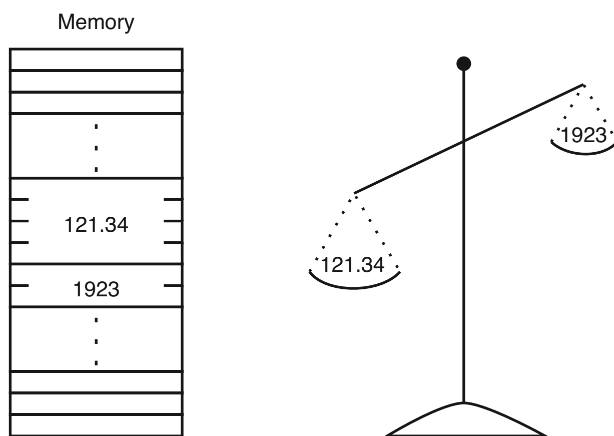


Рис. 2.2. Хранение чисел с плавающей точкой занимает больше памяти, чем хранение целочисленных значений

ПРИМЕЧАНИЕ

На рис. 2.2 мы продемонстрировали вам, что, в общем, целые числа потребляют гораздо меньше памяти, чем числа с плавающей точкой, вне зависимости от величины сохраняемых значений. Большой почтовый короб для посылок может содержать меньше почты, чем маленький. Содержимое гофрокороба не влияет на его возможность хранить определенный объем корреспонденции. На величину числового хранилища языка С влияет не значение числа, а его тип.

В разных компиляторах языка C предусматривается различный объем хранилища целых чисел и чисел с плавающей точкой. Чуть позже вы узнаете, какие существуют способы выяснения точного объема памяти, используемого компилятором для хранения каждого типа данных.

Подведем итог, рассмотрев еще один пример

Целью данной главы было познакомить вас с тем, как «выглядит и ощущается» программа, написанная на языке C и, в первую очередь, конечно же, с функцией `main()`, содержащей выполняемые выражения языка C. Как вы уже узнали, C — язык со свободной формой написания, который не придирается к количеству оставленных вами пробелов. Однако C очень трепетно относится к буквам нижнего регистра. Правила языка C требуют написание имен всех команд и функций, таких как `printf()`, только строчными («маленькими») буквами.

На данном этапе не волнуйтесь по поводу подробностей программного кода, приводимого нами в этой главе: вся оставшаяся книга посвящена объяснению деталей программных кодов. Все же написание и выучивание максимального количества программ было бы хорошей идеей, ведь практика повысит вашу уверенность при написании кода. Поэтому ниже мы приводим еще одну программу, на сей раз использующую все типы данных, которые вы только что изучили:

```
/* Программа, в которой используются символы, целые
числа и числа с плавающей точкой */
#include <stdio.h>
main()
{
  printf("Я изучаю язык программирования %c \n",
'C');
  printf("Я только что дочитал Главу %d\n", 2);
  printf("Я на %.1f процентов готов перейти ", 99.9);
  printf("к следующей главе!\n");
  return 0;
}
```

Приведенная выше программа только лишь выводит три сообщения на экран. Каждое сообщение содержит один из трех изученных нами

в этой главе типов данных: символ (`C`), целое число (`2`) и число с плавающей точкой (`99.9`).

ПРИМЕЧАНИЕ

В первом выражении `printf` команда `%c` указывает программе, куда необходимо вставить символ `'C'`. На самом деле команда `%c` — это сокращение от английского слова «character», которым обозначается символьный тип данных, поэтому она имеет такой вид не потому, что должна вывести на печать символ `'C'`. Если бы изучаемый вами язык назывался языком программирования N, то для вывода символа `'N'` вы все равно бы воспользовались командой `%c`.

В программе программист написал только одну-единственную функцию — функцию `main()`. Код функции `main()` всегда помещается между правой и левой фигурными скобками (`{` и `}`), как, впрочем, и код любой другой функции, которую вы можете добавить в свою программу.

Вы также встретите еще одну функцию: `printf()`, которая является встроенной функцией языка C, выводящей текст на экран.

Ниже приведен вывод, генерируемый программой:

```
Я изучаю язык программирования C
Я только что дочитал главу 2
Я на 99.9 процентов готов перейти к следующей главе!
```

СОВЕТ

Поиграйте с программой, изменяя текст сообщений или данные. Вы также можете специально допустить ошибку при наборе кода, например пропустив точку с запятой (`;`) в конце какой-либо из строк, просто чтобы посмотреть, что в этом случае произойдет при попытке скомпилировать программу. Учитесь на ошибках! Это сделает из вас лучшего программиста!

Абсолютный минимум

Эта глава познакомила вас с тем, как «выглядит и ощущается» программа на C и, в первую очередь, функция `main()`. Ниже приведены основные моменты содержания главы:

- После имени функции в языке C необходимо ставить круглые скобки. Программа на языке C состоит из одной или большего количества функций. Наличие функции `main()` является строго обязательным. Компилятор C выполняет функцию `main()` прежде, чем запустить выполнение любой другой функции в программе.
- Для повышения удобочитаемости добавляйте как можно больше свободного места в текст программного кода
- Не помещайте нули перед целым числом, если это число не есть нуль.
- При использовании символов заключайте их в одинарные кавычки. Строки необходимо помещать в двойные кавычки. Целые числа — это числа без дробной части и дробного разделителя. У чисел с плавающей точкой есть дробный разделитель (точка, а не запятая, как в русском языке. — *Примеч. пер.*).

Глава 3
ЧТО ДЕЛАЕТ ЭТА КОМАНДА? ПОЯСНЕНИЕ КОДА С ПОМОЩЬЮ КОММЕНТАРИЕВ

В этой главе

- Комментирование программного кода
- Вставка комментариев
- Отступы
- Еще один стиль комментариев

Ваш компьютер должен быть в состоянии понимать ваши программы. Однако, так же как любой компьютер, это не более чем тупая железяка, вы должны быть предельно внимательны при наборе команд языка С и употреблять их правильно и строго в той последовательности, в которой они должны исполняться компьютером. Впрочем, возможно, люди тоже будут читать исходные коды ваших программ. Вы часто будете модифицировать свои программы, если же вы будете писать программы для какой-либо компании, то потребности этой компании также могут меняться с течением времени. Вы должны лишний раз убедиться, что ваши программы понятны как *людям*, так и компьютерам. Поэтому вам необходимо документировать создаваемые вами программы, объясняя читателю, что именно они выполняют.

Комментирование программного кода

По тексту программы на С вам следует вставлять различные комментарии — сообщения, разбросанные по тексту программы, объясняющие, что выполняет соответствующий участок кода. Если вы пишете программу для обсчета платежной ведомости, комментарии к коду программы могут пояснять общую выплату заработной платы, подсчет региональных и федеральных налогов, выплаты на соцобеспечение, удержания и отчисления, а также другие необходимые подсчеты.

ПРИМЕЧАНИЕ

Если вы создаете программу для личного использования, вам не очень-то будут нужны комментарии, не так ли? Вообще это утверждение верно лишь отчасти. Язык С сложен синтаксически, поэтому спустя некоторое время после написания программы даже вам, создателю, будет непросто отследить ход ее выполнения, ведь вы можете забыть, зачем написали тот или иной участок кода. Таким образом, комментарии помогут вам расшифровать ранее написанный код.

СОВЕТ

Добавляйте комментарии по ходу написания программ. Приучите себя к этому уже сейчас, так как программисты очень редко возвращаются к написанным программам для вставки комментариев. Впрочем, при возникновении необходимости внести изменения в дальнейшем программисты часто жалуются на недостаток комментариев в коде. У методики вставки комментариев во время написания программы есть еще одно преимущество над вставкой комментариев по окончании работы. Во время написания программы вам часто нужно будет ссылаться на выражения, написанные вами ранее, поэтому вместо повторного разбора написанного вами кода на языке С вы можете просто пробежаться глазами по комментариям, что значительно ускорит поиск нужных вам участков кода. Если же вы не оставляли комментариев, то придется заниматься расшифровкой программного кода на С всякий раз при поиске нужного его участка.

Обслуживание программы — это процесс ее изменения с течением времени для исправления скрытых ошибок и адаптации программы к изменяющейся среде. Если вы пишете программу для расчета платежной ведомости по заказу какой-то коммерческой фирмы, то эта фирма со временем может изменить метод расчета ведомости (например, перейти от расчета два раза в месяц на еженедельный расчет), и вам (или другому программисту) потребуется модифицировать программу, чтобы она соответствовала новым платежным правилам и процедурам фирмы-заказчика. Вставка комментариев убыстряет процесс обслуживания программы. Если в коде есть комментарии, то вы или другой программист сможете просмотреть листинг программного кода для поиска тех участков этого кода, которые нуждаются в изменениях.

Комментарии не являются командами языка С. Компилятор С игнорирует все комментарии, включенные в программу. Комментарии

нужны людям, компьютерам достаются лишь программные выражения, находящиеся вне текстов комментариев.

Взгляните на следующее выражение:

```
return ((s1 < s2) ? s1 : s2);
```

Вы еще не в достаточной степени знакомы с языком C, чтобы понять это выражение, но даже если бы вы хорошо знали этот язык, то вам все равно потребовалось некоторое время, чтобы изучить это выражение и понять, в чем его суть. Не лучше ли, когда оно сопровождено комментарием, как показано ниже:

```
return ((s1 < s2) ? s1 : s2); /* Возвращает наименьшее
из двух чисел */
```

В следующем разделе объясняется синтаксис комментариев, однако уже сейчас можно понять, что выражение между `/*` и `*/` — это и есть комментарий.

Чем ближе текст комментария к разговорному языку и дальше от языка C, тем лучше комментарий. Не следует писать комментарии, просто чтобы они были, так, например, комментарий к следующему выражению абсолютно бесполезен и не нужен:

```
printf("Payroll"); /* Печать слова "Payroll" */
```

ВНИМАНИЕ

Вы еще очень мало знакомы с языком C, но вам *уже* не требуется комментарий к вышеприведенному выражению! Излишнее комментирование — это просто трата вашего времени, ведь ничего полезного в вашу программу они не привнесут. Добавляйте комментарий о выполняемых действиях, которые могут оказаться полезными людям (и вам самим, в том числе), читающим ваш программный код.

Вставка комментариев

В языке C комментарии начинаются с символов `/*` и заканчиваются символами `*/`. Комментарии к программам могут занимать несколько строк и включаться практически в любой участок программного кода. Все приведенные ниже строки кода на языке C содержат комментарии:

```
/* Это пример комментария, занимающего
сразу две строки */
/* Это однострочный комментарий */
for (i = 0; i < 25; i++) /* Считает от 0 до 25 */
```

ПРИМЕЧАНИЕ

Обратите внимание, что комментарии могут занимать целую строку или даже несколько строк, а также могут вставляться перед или после программных инструкций. Выбор места вставки комментария зависит от длины его текста и величины участка кода, который этот комментарий призван описать.

Покерная программа из приложения Б содержит все виды комментариев. Прочитав комментарии к этой программе, вы сможете в общих чертах понять, что делает данная программа, не вчитываясь в сам код на языке С.

Не следует комментировать каждую строку программы. Обычно комментарии требуется вставлять через несколько строк. Многие программисты предпочитают поместить сразу несколько комментариев перед большим участком кода, а затем добавлять небольшие комментарии уже в строках программного кода по мере необходимости. Далее приведена завершенная программа с разными видами комментариев в тексте:

```
/* Первый листинг программного кода из главы 3 of
Руководства по Программированию на С для Новичков
Обучаем программистов писать превосходные программы
с 1994 года! */
/* автор Дин Миллер */
/* Имя файла Chapter3ex1.c */
/* Подсчет суммы расходов на подарки в праздники. */
#include <stdio.h>
main()
{
float gift1, gift2, gift3, gift4, gift5; /* Переменные
для записи расходов */
float total; /* Переменная для записи общей суммы */
/*Запрос стоимости каждого подарка */
```

```
printf("Сколько вы хотите потратить на маму? ");
scanf(" %f", &gift1);
printf("Сколько вы хотите потратить на папу? ");
scanf(" %f", &gift2);
printf("Сколько вы хотите потратить на сестру? ");
scanf(" %f", &gift3);
printf("Сколько вы хотите потратить на брата? ");
scanf(" %f", &gift4);
printf("Сколько вы хотите потратить на своего любимого
");
printf("автора книг по программированию на C? ");
scanf(" %f", &gift5);
total = gift1+gift2+gift3+gift4+gift5; /* Вычисление
общей суммы расходов на все подарки */
printf("\nОбщая сумма, которую вы потратите на
подарки: $%.2f", total);
return 0; /*Конец программы */
}
```

Многие компании требуют, чтобы программисты включали в верхнюю часть программного кода свои имена. Если в будущем потребуется изменить программу, то можно будет найти ее создателя с тем, чтобы он оказал помощь в ее изменении. Также хорошей практикой программирования является написание имени, под которым вы сохранили файл исходного кода на диске, это поможет найти нужный файл, если вы читаете программный код в распечатанном на бумаге виде.

ПРИМЕЧАНИЕ

В этой книге, особенно в первых главах, может показаться, что мы слишком дотошно комментируем программный код. Но мы посчитали, что вы еще очень плохо знакомы с языком C, и каждая маленькая подсказка может быть полезной.

СОВЕТ

С целью тестирования иногда бывает полезно закомментировать участок кода, то есть поместить его между символами `/*` и `*/`. Сделав так, вы заставите компилятор C проигнорировать этот участок

кода, чтобы вы могли сконцентрироваться на работе с другим его участком. Однако ни в коем случае нельзя закомментировать участок кода, уже содержащий комментарии, так как комментарии в C не могут вкладываться один в другой. Первое же сочетание символов `*/` будет сигнализировать компилятору окончание комментария, когда же компилятор найдет следующее сочетание `*/` без соответствующего сочетания `/*`, вы получите сообщение об ошибке.

Отступы

Отступы — это сочетание пробельных символов и пустых строк, которые вы можете встретить в большинстве программ. В некотором роде, отступы гораздо важнее комментариев, так как они делают ваши программы более удобочитаемыми. При чтении программных кодов, написанных на языке C, людям необходимы отступы в этих текстах, так как благодаря им удобочитаемость таких программ значительно выше по сравнению с программами, инструкции в которых написаны слишком близко друг к другу. Рассмотрим следующую программу:

```
#include <stdio.h>
main(){float a, b; printf("Какой бонус вы получили?
");scanf(" %f",
&a); b = .85  * a; printf("Если отдадите 15 процентов
на благотворительность, у вас все еще останется
%.2f.", b);retrurn0;}
```

Для компилятора C вышеприведенная программа является идеальной. Однако у вас ее чтение может вызвать головную боль. Несмотря на то, что код программы очень прост и выяснение выполняемой им задачи не требует долгих усилий, программный код ниже *значительно* проще расшифровать, хотя в нем отсутствуют комментарии:

```
#include <stdio.h>
main()
{
float a, b;
printf("Какой бонус вы получили?");
scanf(" %f", &a);
b = .85  * a;
```

```
printf("Если отдадите 15 процентов на
благотворительность,.");
printf("у вас все еще останется %.2f ", b);
retrurn0;
}
```

Вышеприведенный листинг программного кода идентичен предыдущему, с той лишь разницей, что он включает отступы и разрывы строк. Физическая длина программы не определяет удобочитаемость, удобочитаемость определяется количеством отступов. (Конечно, несколько комментариев также улучшили бы удобочитаемость этой программы, но цель упражнения — показать вам разницу между отсутствием отступов и нормальным количеством отступов).

ПРИМЕЧАНИЕ

Вы можете задаться вопросом, почему в первой строке сжатого программного кода, той самой, которая содержала инструкцию `#include`, после закрывающей угловой скобки не было никакого кода, ведь, в конце концов, программа, в которой после строки `#include` был бы помещен еще какой-то код, стала бы еще менее удобочитаемой. Дело в том, что Code::Blocks (а также некоторые другие компиляторы) не разрешают вставлять код после директивы `#include` (или любой другой программной инструкции, начинающейся со знака «решетка» [#]).

Еще один стиль комментариев

Современные компиляторы С поддерживают еще один вид комментариев, которые были изначально разработаны для языка С++, выпустив стандарт С99, комитет Американского национального института стандартизации (ANSI) принял новый стиль комментариев, поэтому вы можете спокойно им пользоваться (разумеется, кроме случаев, когда для написания программ вы используете очень-очень старый компьютер и компилятор). Комментарии этого стиля начинаются с двух слешей (//) и заканчиваются только в конце строки. Ниже показан пример комментария нового стиля:

```
//Еще один пример кода с другим стилем комментариев

#include <stdio.h>
```

```
main()
{
  printf("I like these new comments!"); //Простое
выражение
  return 0;
}
```

Оба стиля комментариев работают нормально, поэтому в примерах программ в этой книге используются оба стиля по мере необходимости. Мы рекомендуем вам принять во внимание оба стиля комментариев, так как в более сложных программах оба стиля окажутся вам полезными.

Абсолютный минимум

Вы должны комментировать свои программы не для компьютеров, но для людей. Программы на языке C могут быть криптологически сложны для восприятия, а комментарии помогают избежать двусмысленности и непонимания. Ниже перечислены основные ключевые моменты, которые необходимо запомнить по прочтении этой главы.

- Три правила программирования: комментировать, комментировать и еще раз комментировать. Сделайте ваш код понятным благодаря изобилию комментариев.
- При вставке многострочных комментариев начните их с символов /* — и компилятор языка C будет воспринимать весь текст, следующий за этими символами, как комментарий, до тех пор, пока он не встретит закрывающие символы */.
- При вставке однострочных комментариев вы можете использовать символы //. Компилятор C игнорирует всю строку текста, следующего за этими символами.
- Для повышения удобочитаемости программ используйте отступы и разрывы строк.

Глава 4

МИРОВАЯ ПРЕМЬЕРА: ВЫХОД ВАШЕЙ ПРОГРАММЫ НА ЭКРАНЫ

В этой главе

- Использование `printf()`
- Печать строк
- Кодирование управляющих последовательностей
- Использование символов преобразования
- Обобщение и пример программного кода

Ваша программа вряд ли может считаться полезной, если ни вы, ни другие люди не будут видеть на экране ее вывод, в конце концов, вы же должны видеть результаты ее работы. Основное средство вывода данных на экран в языке С — это функция `printf()`. В С нет специальных команд для организации вывода данных на экран, но функция `printf()` входит в состав всех компиляторов языка С и является одной из самых часто используемых функций языка.

Использование функции printf()

Основная задача функции `printf()` — вывод данных на экран компьютера. Как было показано в примерах программ в главах 2 и 3, функция `printf()` посылает на экран символы, числа и слова. Функция `printf()` обладает богатыми возможностями, но вам не нужно быть экспертом по применению этой функции (на самом деле очень немногих программистов можно назвать экспертами по `printf()`), чтобы использовать ее для организации вывода на экран всех выходных данных вашей программы.

Формат функции printf()

Функция `printf()` может принимать большое количество форм, но, привыкнув к ее формату, вы найдете ее очень легкой в использовании.

Ниже приведет общий формат функции `printf()`:

```
printf(контрольная строка [, данные]);
```

По тексту этой книги при первом знакомстве с какой-либо функцией или командой мы всегда будем приводить ее базовый формат. Формат — это некий общий вид какого-либо выражения. Если что-либо в формате выражения приводится в квадратных скобках, как, например, элемент `, данные` в функции `printf()`, это значит, что этот элемент является необязательным. Вводить с клавиатуры квадратные скобки практически никогда не нужно. Если для какой-либо команды квадратные скобки необходимы, это будет отдельно указано в тексте, описывающем эту команду. Таким образом, функция `printf()` требует *`контрольную строку`*, но данные, следующие после нее, являются необязательным элементом.

ВНИМАНИЕ

На самом деле функция `printf()` не посылает никаких выходных данных на экран, вместо этого она отправляет их на стандартное устройство вывода вашего компьютера. Большинство операционных систем, в том числе Windows, перенаправляют стандартный вывод на экран компьютера, конечно же, если вы, зная, как это сделать, не направите выходные данные на какое-то другое устройство вывода. В большинстве случаев вы можете не заботиться об установке стандартного устройства вывода, так как чаще всего будете выводить информацию из программы именно на экран компьютера. Другие функции C, с которыми вы познакомитесь чуть позже, направляют выходные данные на принтер и дисководы.

ПРИМЕЧАНИЕ

Вы можете задаться вопросом, почему некоторые слова в строке формата функции напечатаны курсивом. Так сделано потому, что они являются всего лишь «заглушками». Заглушка представляет собой имя, символ или формулу, которую вы передаете функции. При указании стандартных форматов функций и команд заглушки печатаются курсивным шрифтом, чтобы вы могли понять, что вместо них надо подставить соответствующие данные.

Ниже приведен пример функции `printf()`:

```
printf("Мой любимый номер %d", 7); //Выводит на экран
мое любимое
// число 7
```

Как мы уже упоминали ранее (в главе 2), все строки текста в C должны быть заключены в кавычки, поэтому *контрольная строка* также заключается в кавычки. Все элементы, следующие за *контрольной строкой*, не являются обязательными и определяются значениями, которые вы хотите вывести на экран.

ПРИМЕЧАНИЕ

После всех команд и функций согласно правилам языка C должны следовать точка с запятой (`;`), сообщающие компилятору, что строка с программной инструкцией завершена. После фигурных скобок и первых строк функций точка с запятой не нужны, так как эти строки не содержат никаких выполняемых программных инструкций. Все выражения с `printf()` должны заканчиваться точкой с запятой, однако после `main()` точка с запятой не нужны, так как вы не даете эксплицитную (явную) команду компилятору C выполнить функцию `main()`. Однако вы даете команду компилятору C выполнить функцию `printf()` и большое количество других функций. По мере изучения языка C вы получите дополнительную информацию по правильному употреблению знака точка с запятой.

Печать строк

Строки сообщений — это самый простой из типов данных, которые можно распечатать при помощи функции `printf()`. Все, что вам нужно — заключить строку с сообщением в двойные кавычки. Указанная в примере ниже функция `printf()` выводит текст сообщения на экран:

```
printf("Вы на пути к овладению языком C");
```

Сообщение `Вы на пути к овладению языком C` появится на экране, когда компьютер начнет выполнение программы.

ПРИМЕЧАНИЕ

Для функции `printf()` строка `Вы на пути к овладению языком C` является *контрольной строкой*, хотя в этом примере вряд ли можно увидеть какой-то контроль, т.к. он подразумевает просто вывод сообщения на экран.

Результатом выполнения следующих двух выражений `printf()` станет не вполне ожидаемое сообщение на экране:

```
printf("Пишите код");
printf("Изучайте С");
```

Вот, что выведут на экран эти два выражения `printf()`:

```
Пишите кодИзучайте С
```

СОВЕТ

В языке С не предусмотрен автоматический перевод курсора на следующую строку при выполнении функции `printf()`. Если вы хотите перейти на следующую строку, вам необходимо будет ввести в *контрольную строку* специальную управляющую последовательность.

Управляющие последовательности

В языке С предусмотрено большое количество управляющих последовательностей, некоторые из них вы будете использовать практически во всех своих программах. В табл. 4.1 приводится перечень некоторых наиболее часто используемых управляющих последовательностей.

Табл. 4.1 Управляющие последовательности

Код	**Описание**
`\n`	Новая строка
`\a`	Звуковое оповещение (через динамик компьютера)
`\b`	Возврат каретки на один символ (**Backspace**)
`\t`	Табуляция
`\\`	Вывод символа «\» (обратный слеш)
`\'`	Вывод апострофа (одинарной кавычки)
`\"`	Вывод двойной кавычки

ПРИМЕЧАНИЕ

Термин «*управляющая последовательность*» звучит сложнее, чем он есть на самом деле. Управляющая последовательность в языке С представляет собой единичный символ, производящий эффект, описанный в таблице 4.1. Когда компилятор С пытается вывести на экран символ '`\a`', то вместо печати на экране символов «\» и «a», компьютер издает звуковой сигнал.

В функции `printf()` вы встретите большое количество управляющих последовательностей. Каждый раз, когда вам нужно «перейти вниз» на следующую строку при выводе текста на экран, вы должны употребить управляющую последовательность `\n`, таким образом компилятор C создаст новую строку, на которую переведет мигающий курсор. Следующие два выражения `printf()` выводят свои сообщения на разных строках благодаря управляющей последовательности `\n`, указанной в контрольной строке первой функции:

```
printf("Пишите код\n");
printf("Изучайте C");
```

СОВЕТ

Мы могли бы поместить управляющую последовательность `\n` и в начало второй строки — и в результате также получили бы две строки текста на экране. Так как для компилятора C управляющие последовательности являются отдельными символами, их необходимо заключать в кавычки, чтобы компилятор C понял, что эти символы являются частью сообщения, выводимого на экран. Следующее выражение также выводит на экран две строки текста:

```
printf("Пишите код\nИзучайте C");
```

Символы заключаются в одинарные кавычки, начало и конец строк символов обозначаются двойными кавычками, обратный слеш обозначает начало управляющей последовательности. Таким образом, для вывода на экран кавычек (как одинарных, так и двойных) вам необходимо будет воспользоваться соответствующими управляющими последовательностями.

Последовательность `\a` извлекает звук из динамика (бипера) вашего компьютера, `\b` — смещает курсор в обратном направлении, а `\t` позволяет вывести данные, сдвинув их на несколько пробелов. В C поддерживаются также и другие управляющие последовательности, но пока что именно этими вы будете пользоваться чаще всего.

Пример программного кода, приведенный в листинге далее, демонстрирует использование всех управляющих последовательностей, перечисленных в табл. 4.1. Как обычно, будет лучше, если сначала вы попробуете скомпилировать эту программу в том виде, в котором мы приводим ее здесь, а затем измените ее так, как вам хочется:

```
// Путеводитель по C для новичков, издание третье
// Глава 4 Пример 1-- Chapter4ex1.c
#include <stdio.h>
main()
{
    /* Эти три строки демонстрируют использование
самых популярных управляющих последовательностей */
    printf("Column A\tColumn B\tColumn C");
    printf("\nMy Computer\'s Beep Sounds Like This:
\a!\n");
    printf("\"Letz\bs fix that typo and then show the
backslash ");
    printf("character \\\" she said\n");
    return 0;
}
```

После того как вы введете, скомпилируете и запустите программу, вы получите следующие результаты:

```
Column A    Column B    Column C
My Computer's Beep Sounds Like This: !
"Let's fix that typo and then show the backslash
character \" she said
```

ПРИМЕЧАНИЕ

Считаем необходимым прояснить несколько нюансов предыдущего программного кода. Первое — всегда помещайте строку `#include <stdio.h>` в начало всех программ, в которых вы используете функцию `printf()`, так как эта функция определена в файле `stdio.h`. Если вы забудете включить эту строку в программный код, компилятор выдаст сообщение об ошибке, так как ваша программа не будет знать, как следует выполнять инструкцию `printf()`. Кроме того, разные компиляторы языков C/C++ могут выводить различное количество символов пробела при интерпретации управляющей последовательности `\t`. Наконец, важно помнить, что управляющая последовательность `\b` переписывает предыдущее содержание знакоместа, поэтому в примере буква `'z'` не выводится на экран, вместо нее мы видим только букву `'s'`.

Символы преобразования

При выводе символов и чисел необходимо сообщать компилятору С, каким образом нужно вывести их на экран. Символы преобразования позволяют указать формат выводимых чисел. В табл. 4.2 приведены несколько наиболее часто используемых символов преобразования языка С.

Табл. 4.2 «Символы преобразования»

Символ преобразования	Описание
`%d`	Целое число
`%f`	Число с плавающей точкой (дробь)
`%c`	Символ
`%s`	Строка

Если вы хотите вывести какое-либо значение внутри строки, вставьте в *контрольную строку* соответствующие символы преобразования. Далее, правее *контрольной строки*, перечислите значения, которые вы хотите напечатать. На рис. 4.1 изображен пример того, как функция `printf()` распечатывает три числа: целое, дробное (с плавающей точкой) и еще одно целое.

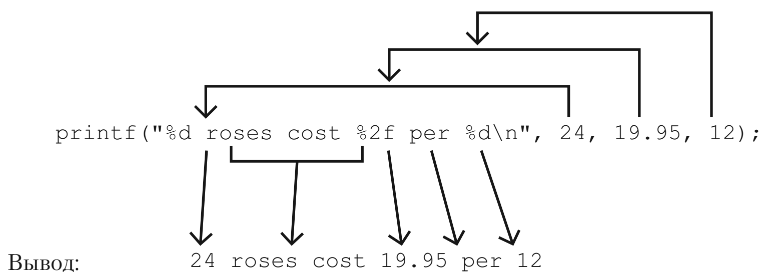


Рис. 4.1. Символы преобразования функции `printf()` определяют, как и где следует выводить числа

Для символов и строк также предусмотрены собственные символы преобразования. Хотя вам может и не понадобиться символ `%s` для вывода строк самих по себе, но вы можете использовать его для вывода строк в сочетании с данными. Следующий пример функции `printf()` выводит на экран разные типы данных, используя для этого соответствующие символы преобразования:

```
printf("%s %d %f %c\n", "Sam", 14, -8.76, 'X');
```

Приведенная выше функция printf() производит следующий вывод:

```
Sam 14 -8.760000 X
```

ПРИМЕЧАНИЕ

Строке `Sam` необходимы двойные кавычки, а для символа `X`, как и для прочих символов, нужно использовать одинарные кавычки.

ВНИМАНИЕ

У языка С есть одно странное свойство: крайняя точность по отношению к десятичным дробям. Несмотря на то, что число –8.76 содержит в дробной части только два числа, язык С настойчиво выведет на экран шесть чисел дробной части.

Однако вы можете контролировать, каким образом С выводит на экран числа с плавающей точкой. Для этого между символами `%` и `f` необходимо вставить точку (`.`) и число. Указанное вами число определит количество чисел после запятой, которое нужно вывести на экран. В следующем примере функция `printf()` выводит на экран четыре разных числа, несмотря на то, что этой функции задается одно и то же число:

```
printf("%f  %.3f  %.2f  %.1f", 4.5678, 4.5678, 4.5678,
4.5678);
```

Язык С округляет числа с плавающей точкой до количества чисел дробной части, указанного в символе преобразования `%.f` и производит следующий вывод:

```
4.567800  4.568  4.57  4.6
```

СОВЕТ

Сейчас вы, возможно, еще не можете оценить значимость символов преобразования, так как вы, наверное, догадались, что всю информацию можно было бы просто включить в *`контрольную строку`*. Однако вы поймете всю ценность символов преобразования, познакомившись при чтении следующей главы с переменными.

`Контрольная строка` функции `printf()` полностью соответствует внешнему виду экранного вывода функции. Единственная причина,

по которой выведенные на экран числа разделяют два пробела — в том, что в *контрольной строке* символы %f также разделяются двумя пробелами.

Обобщение и пример программного кода

Рассмотрим следующий листинг программного кода.

```
/* Путеводитель по С для новичков, издание третье
Глава 4 Пример 2--Chapter4ex1.c */
#include <stdio.h>
main()
{
    /* Вот еще код, помогающий вам освоить Функцию
printf(), Управляющие последовательности и Символы
преобразования */
    printf("Quantity\tCost\tTotal\n");
    printf("%d\t\t$%.2f\t$%.2f\n", 3, 9.99, 29.97);
    printf("Too many spaces     \b\b\b\b can be fixed
with the ");
    printf("\\%c Escape character\n", 'b');
    printf("\n\a\n\a\n\a\n\aSkip a few lines, and beep
");
    printf("a few beeps.\n\n\n");
    printf("%s %c.", "You are kicking butt learning",
'C');
    printf("You just finished chapter %d.\nYou have
finished ", 4);
    printf("%.1f%c of the book.\n", 12.500, '%');
    printf("\n\nOne third equals %.2f or ", 0.333333);
    printf("%.3f or %.4f or ", 0.333333, 0.333333);
    printf("%.5f or %.6f\n\n\n", 0.333333, 0.3333333);
    return 0;
}
```

Введите вышеприведенный код, скомпилируйте и запустите программу. То, что вы увидите на экране, показано на рис. 4.2.

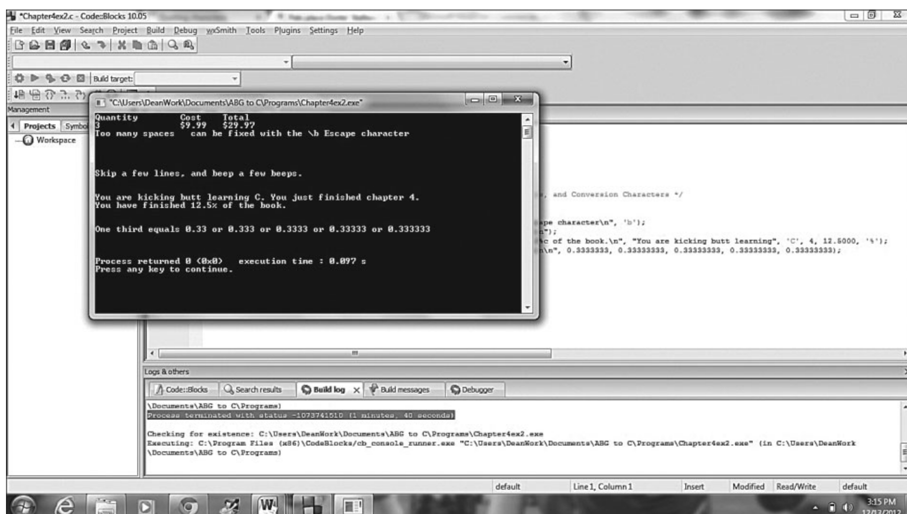


Рис. 4.2. Экранный вывод второго листинга главы 4

Обратите внимание, что из-за длины слова «`Quantity`» (количество) во второй строке потребовалось употребить табуляцию дважды, чтобы стоимость позиции оказалась точно под заголовком «`Cost`» (стоимость). Возможно, это вам не понадобится, так как вам нужно будет лишь протестировать код, чтобы понять, на сколько символов пробела передвигается курсор при использовании управляющей последовательности табуляции. Иногда пропуска только одной строки может оказаться недостаточно.

К счастью, мы можем употребить столько символов `\n`, на сколько строк вниз нам нужно перевести курсор. Наконец, поняв, что знак `%` является важной частью символов преобразования, вы не можете просто ввести его в *`контрольную строку`* и ожидать, что программа его напечатает. Поэтому, если вам необходимо вывести на экран знак процента, используйте символ преобразования `%c` в том месте, где его будет нужно заменить на `%`.

Абсолютный минимум

Программы, создаваемые вами, должны уметь общаться с пользователем, сидящим за клавиатурой. Функция `printf()` выводит данные на экран. Ниже приведены основные моменты текста этой главы, которые нужно запомнить:

- Функции `printf()` необходима контрольная строка, определяющая то, каким образом будет выведены на печать ваши данные.
- Не ждите, что язык С автоматически отформатирует ваши данные. Для этого необходимо использовать символы преобразования.
- Для разрыва строк, табуляции, вывода кавычек и обратных слешей, а также подачи звукового оповещения воспользуйтесь соответствующими управляющими последовательностями
- Если вы не хотите, чтобы десятичная дробь выводилась на экран с шестью знаками после запятой (точки), воспользуйтесь символом %f, позволяющем получить контроль над дробной частью натурального числа.

Глава 5
ДОБАВЛЕНИЕ ПЕРЕМЕННЫХ В ПРОГРАММУ

В этой главе

- Идентификация типов переменных
- Именование переменных
- Определение переменных
- Сохранение данных в переменных

Без сомнений, вы слышали, что компьютеры обрабатывают данные, поэтому должен быть какой-то способ сохранения этих данных. В C, как и в большинстве языков программирования, для хранения данных используются *переменные*. Переменная представляет себя коробочку в памяти компьютера, содержащую число или символ. В главе 2 мы рассказывали вам про разные типы данных: символы, строки, переменные и числа с плавающей точкой. В этой главе мы объясним вам, как можно сохранять в программе вышеперечисленные типы данных.

Виды переменных

Из-за существования нескольких видов данных в C предусмотрено несколько видов переменных. Будет ошибочным утверждать, что любая переменная может хранить любые данные. Только целочисленные переменные могут хранить целые числа, и только вещественные переменные могут хранить числа с плавающей точкой, и так далее.

ПРИМЕЧАНИЕ

Читая эту главу, образно представляйте переменные в виде почтовых коробов в местном отделении почтовой связи. Почтовые гофрокороба отличаются по размеру, и на каждом из них нанесен уникальный номер — почтовый идентификатор. Создаваемые вами программы на языке C также отличаются по размеру, в зависимости

от того, какого вида данные они содержат. Каждая переменная обладает уникальным именем, отличающим ее от других переменных.

Типы данных, которые вы изучили в главе 2, называются *литеральными данными* (или иногда *постоянными (константными) данными*), так как конкретные числа и буквы не меняются. Число `2` и символ `'x'` всегда остаются числом `2` и символом `'x'`. Большое количество данных, с которыми вам придется работать, например возраст, величина заработной платы и вес, непостоянны и изменяются. Если бы вы писали программу-обработчик платежных ведомостей, то вам потребовался бы некий механизм для сохранения изменяющихся данных. Таким механизмом являются переменные. Переменные — это нечто большее, чем коробки в памяти, которые могут хранить изменяющиеся с течением времени данные.

Существует большое количество различных типов данных. В табл. 5.1 мы приводим несколько наиболее часто используемых типов. Обратите внимание, что большинство переменных относятся к одному из типов данных (символ, целое число или число с плавающей точкой), аналогично литеральным данным. В конце концов, вам же нужно место для хранения целых чисел, его предоставляют вам целочисленные переменные.

Табл. 5.1. Некоторые наиболее часто используемые типы переменных языка С

Название	Описание
`char`	Хранит символьные данные, такие как 'X' и '*'
`int`	Хранит целочисленные данные, например, `1`, `-32` и `435125`. Может хранить данные в диапазоне значений от `-2147483648` и `2147183647`
`float`	Хранит данные в виде чисел с плавающей точкой, такие как `25.6`, `-145.23` и `.000005`
`double`	Хранит сверхбольшие или сверхмалые числа с плавающей точкой

СОВЕТ

В некоторых старых компиляторах языка С тип данных `int` мог хранить числа в диапазоне от `-32768` до `32767`. Если вам нужно было работать с большими числами, то вам было необходимо пользоваться типом данных `long int`. Однако в большинстве современных компиляторов тип данных `int` может хранить значения в том же самом диапазоне, что и `long int`. Если вы хотите удостовериться в возможностях вашего компилятора, можете воспользоваться оператором `sizeof`, речь о котором пойдет в главе 13.

ВНИМАНИЕ

Возможно, вы обратили внимание на отсутствие строковых переменных, несмотря на *существование* литерального типа данных «строка символов». Язык C — один из немногих языков программирования, не поддерживающих строковых переменных, но, как вы убедитесь в главе 6, все же существует способ хранить строки в переменных.

В столбце «Название» табл. 5.1 перечислены ключевые слова, необходимые при создании переменных в программе. Иными словами, если вы хотите создать целочисленную переменную, вам потребуется ключевое слово `int`. Прежде чем вы закончите изучение переменных и перейдете к их использованию, вам потребуется знать еще одно: как именовать переменные.

Именование переменных

У всех переменных есть имена и, поскольку именно вы ответственны за их именование, вам необходимо выучить несколько правил. Все имена переменных должны отличаться: в программе не может быть двух переменных с одинаковыми именами.

Имя переменной может состоять из символов в количестве от 1 до 31. Несмотря на то, что некоторые компиляторы позволяют создание более длинных имен переменных, лучше все-таки придерживаться этого ограничения для повышения переносимости кода и минимизации количества опечаток, ведь чем длиннее имя, тем больше вероятность опечатки. Имя переменной должно начинаться с буквы латинского алфавита, после которой могут следовать другие буквы латинского алфавита, цифры и знаки подчеркивания в любой последовательности. Ниже приведены примеры правильных имен переменных:

```
myData pay94  age_limit    amount QtlyIncome
```

СОВЕТ

Язык C допускает, что имена переменных могут начинаться с символа подчеркивания, однако вам не следует так поступать. Дело в том, что некоторые встроенные переменные языка C начинаются с подчеркивания, поэтому всегда есть вероятность, что вы перекроете имя встроенной переменной, если имя вашей переменной также начинается

с символа подчеркивания. Лучше пойти безопасным путем и всегда начинать имена переменных с буквы латинского алфавита. Мы еще раз подчеркиваем важность этого тезиса! (Ну вы понимаете…)

Ниже приведены примеры *неправильных* имен переменных:

```
94Pay  my Age lastname,firstname
```

Вы должны научиться понимать, почему имена вышеприведенных переменных неверны. Имя первой переменной начинается с цифры, второй переменной, `my  Age`, содержит пробел, а третьей, `lastname,firstname`, — специальный символ (`,`).

ВНИМАНИЕ

Не присваивайте переменной имя функции или команды. Если вы дадите переменной имя команды, то ваша программа попросту не запустится, если же вы дадите переменной имя функции, то дальнейшее использование функции с этим именем вызовет ошибку.

Объявление переменных

Прежде чем использовать переменную, ее необходимо *объявить*. Объявление переменной (иногда еще называется *декларацией переменной*) — не что иное, как информирование компилятора языка C о том, что вам потребуется зарезервировать небольшой участок переменной памяти. Чтобы объявить переменную, вам нужно указать ее тип и имя. Ниже приведены несколько первых строк программы, в которой используются переменные:

```
main()
{
        // Мои переменные для программы
        char answer;
        int quantity;
        float price;
/* Остальной код программы ниже */
```

Образец кода, приведенный выше, содержит три переменных: `answer`, `quantity` и `price`. Они могут хранить три различных вида

данных: символьные данные, целочисленные данные и данные в виде числа с плавающей точкой. Если бы в программе эти переменные не были определены, то мы не смогли бы использовать их для хранения данных.

В одной строке можно объявить сразу несколько переменных одного типа данных. Например, если бы вам нужно было объявить две символьных переменных, а не одну, вы могли бы поступить следующим образом:

```
main()
{
       // Мои переменные для программы
       char first_initial;
       char middle_initial;
/* Остальной код программы ниже */
```

или так:

```
main()
{
       // Мои переменные для программы
       char first_initial, char middle_initial;
/* Остальной код программы ниже */
```

СОВЕТ

Определение большинства переменных осуществляется после открывающейся фигурной скобки сразу после имени функции. Такие переменные называются *локальными переменными*. Язык C позволяет создавать *глобальные* переменные, которые необходимо определять до имени функции, например, до имени функции `main()`. В большинстве случаев предпочтительно использование локальных переменных вместо глобальных . Глава 30 посвящена различиям между локальными и глобальными переменными. Однако сейчас в своих программах просто придерживайтесь использования локальных переменных.

Сохранение данных в переменных

С помощью *оператора присваивания* значения записываются в переменные. Звучит это действие гораздо сложнее, чем есть на практике. Оператор присваивания — это просто знак равенства (=). Синтаксис операции записи значения в переменную выглядит так:

переменная = данные;

Элемент *переменная* — это имя переменной, в которую вы хотите сохранить данные. Переменная должна быть заранее объявлена, как показано в предыдущем разделе. Элемент *данные* может быть числом, символом или математическим выражением, результатом выполнения которого является число. Ниже — пример выражений, присваивающих значения переменным, объявленным в предыдущем разделе:

```
answer = 'B';
quantity = 14;
price = 7.95;
```

В переменной также можно сохранить ответы на математические выражения:

```
price = 8.50 * .65; // Вычисляет цену с 35% скидкой
```

В выражении вы можете использовать другие переменные:

```
totalAmount = price * quantity; /* Использует значения
других переменных */
```

СОВЕТ

Знак равенства говорит компилятору примерно следующее: «Возьми то, что справа, и прилепи это к переменной слева». Знак равенства в данном случае выступает в роли стрелки-указателя влево, как бы говорящей «Вон туда!» Да, в числах никогда не используйте запятые, вне зависимости от того, насколько длинны эти числа!*

Давайте используем полученные вами знания о переменных для создания программы. Откройте редактор, введите, скомпилируйте и запустите следующую программу:

```
// Программа-пример №1 из Главы 5 Руководства для
начинающих
// по С, 3-е издание
```

* В англоговорящих странах принято разделять разряды числа больше тысячи запятыми, например число, записанное как 1,369, отнюдь не является «одной целой тремястами шестьюдесятью девятью десятитысячными», а запись числа «один миллион двести тысяч» будет выглядеть как 1,200,000. Однако такая запись в языке С является неприемлемой, поэтому вышеприведенные числа в программах на С будут иметь следующий вид: 1369 и 1200000 соответственно. — *Примеч. пер.*

```
// Файл Chapter5ex1.c
/* Это пример программы, перечисляющей три типа
школьных принадлежностей, а также стоимость их покупки
*/
#include <stdio.h>
main()
{
    // Объявление и определение некоторых переменных
    char firstInitial, middleInitial;
    int number_of_pencils;
    int number_of_notebooks;
    float pencils = 0.23;
    float notebooks = 2.89;
    float lunchbox = 4.99;
    //Информация для первого ребенка
    firstInitial = 'J';
    middleInitial = 'R';
    number_of_pencils = 7;
    number_of_notebooks = 4;
    printf("%c%c needs %d pencils, %d notebooks, and 1
lunchbox\n",
           firstInitial, middleInitial,number_of_
pencils,
           number_of_notebooks);
    printf("The total cost is $%.2f\n\n", number_of_
pencils*pencils
    + number_of_notebooks*notebooks + lunchbox);
    //Информация для второго ребенка
    firstInitial = 'A';
    middleInitial = 'J';
    number_of_pencils = 10;
    number_of_notebooks = 3;
    printf("%c%c needs %d pencils, %d notebooks, and 1
lunchbox\n",
```

```
           firstInitial, middleInitial,number_of_
pencils,
           number_of_notebooks);
    printf("The total cost is $%.2f\n\n", number_of_
pencils*pencils
    + number_of_notebooks*notebooks + lunchbox);
    //Информация для третьего ребенка
    firstInitial = 'M';
    middleInitial = 'T';
    number_of_pencils = 9;
    number_of_notebooks = 2;
    printf("%c%c needs %d pencils, %d notebooks, and 1
lunchbox\n",
           firstInitial, middleInitial,number_of_
pencils,
           number_of_notebooks);
    printf("The total cost is $%.2f\n",
number_of_pencils*pencils + number_of_
notebooks*notebooks + lunchbox);
    return 0;
}
```

В программе приводятся примеры именования и определения переменных различного типа, а также присваивания им значений. Важно отметить, что вы можете использовать переменную повторно, просто присвоив ей новое значение. Вы, возможно, уже задались вопросом, зачем использовать изначально, а потом еще раз повторно переменные, если вы можете просто изменить значение в самом коде? Почему бы не обойтись без переменных и просто не использовать нужные значения непосредственно там, где они нужны? Значимость переменных станет вам более ясна после прочтения главы 8, когда вы сможете получать значения для переменных непосредственно от пользователя.

Покерная программа из приложения Б должна следить за большим количеством нюансов игры, поэтому в ней используется большое количество переменных. В начале почти каждой функции программы вы можете видеть участок кода, отвечающий за определение переменных.

СОВЕТ

Вы можете определять переменные и присваивать им начальные значения одновременно. Предыдущая программа присваивает значения переменным с плавающей точкой `pencil`, `notebook` и `lunchbox` сразу при их объявлении.

Абсолютный минимум

Эта глава была посвящена различным типам переменных в языке С. Из-за того, что существуют различные виды данных, в С предусмотрены различные виды переменных для хранения этих данных. Ниже перечислены основные моменты текста главы.

- Запомните правила именования переменных, так как вы будете их использовать практически во всех своих программах.
- Всегда определяйте переменные перед использованием.
- Не путайте типы данных и типы переменных, потому что иначе вы получите неправильные результаты.
- При необходимости вы можете определить больше одной переменной в одной строке
- Не используйте запятую для разделения разрядов числа. Вводите число «сто тысяч» как `100000`, а не `100,000`.
- При сохранении значений в переменных используйте знак равенства (=), также называемый оператором присваивания.

Глава 6
ДОБАВЛЕНИЕ СЛОВ В ПРОГРАММУ

В этой главе

- Символ конца строки
- Определение длины строки
- Использование массивов символов: перечисление символов
- Инициализация строк

Несмотря на то, что в языке С нет строковых переменных, у вас все же есть способ сохранения строковых данных. Глава объясняет, каким образом это можно сделать. Вы уже знаете, что строковые данные нужно заключать в кавычки. Даже одиночный символ, заключенный в двойные кавычки, является строкой. Вы также знаете, как выводить строки на печать с помощью функции `printf()`.

Теперь осталось лишь узнать, как использовать специальную разновидность символьной переменной для хранения строковых данных, чтобы ваша программа могла вводить, обрабатывать и выводить на экран строковые данные.

Символ конца строки

В языке С со строками происходит множество странных вещей, например, в конце каждой строки С добавляет нуль. У нуля в конце строки есть несколько названий:

- Нуль-символ
- Бинарный нуль
- Символ конца строки
- ASII 0
- `\0`

ВНИМАНИЕ

Единственное, как вы *не можете* обозначить нуль-символ конца строки, так это «*нуль*»! Программисты, работающие с языком С, используют специальные названия для обозначения нуль-символа конца строки, таким образом, чтобы вы знали, что обычный числовой нуль или символ `'0'` не используются в конце строки. В конце строки может использоваться только специальный *нуль-символ*.

В С завершающий нуль-символ используется для обозначения конца всех строк. Вам никогда не нужно будет указывать его дополнительно при введении какого-либо строковой константы, например фразы `"Меня зовут Юля"`, так как язык С сделает все за вас. Вы никогда не увидите нуль-символ, но он всегда присутствует в памяти. Компилятор С понимает, что достиг конца строки, только когда он находит нуль-символ.

ПРИМЕЧАНИЕ

Приложение А «Таблица ASCII» содержит таблицу ASCII (о которой мы уже говорили в главе 2 «Ваша первая программа на С»). Самый первый символ обозначен как *null*, а его номер ASCII 0. Просмотрите по таблице вниз до строки ASCII 48 — и вы найдете 0. Символ ASCII 48 — это символ `'0'`, тогда как первый символ таблицы ASCII — это *нуль-символ*. С автоматически помещает нуль-символ в конце строки, даже такая короткая строка как `"Мне уже 20"` заканчивается на ASCII 0, который стоит сразу же за символом `0` в числе `20`.

Иногда нуль-символ называют `\0` (*обратный слеш-нуль*), так как вы можете представить нуль-символ, заключив `\0` в одинарные кавычки. Поэтому `'0'` — это символьный нуль, а `'\0'` — это символ конца строки. (Вспомните управляющие последовательности, изученные нами в главе 4 «Мировая премьера: выход вашей программы на экраны», которые также являются одиночными символами, представляемыми двумя символами: обратным слешем и буквой или другим символом. Теперь вы можете добавить в коллекцию еще и обратный слеш с цифрой.)

На рис. 6.1 показано, как строка `"Чудик"` сохранена в памяти компьютера. Как вы видите, эта строка занимает в памяти 6 байт (*байт* — это один участок памяти), несмотря на то, что в строке всего пять символов. Еще один из шести участков памяти занимает нуль-символ, также являющийся частью строки `"Чудик"`.

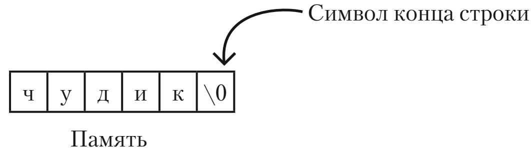


Рис. 6.1. Строка всегда заканчивается на нуль-символ, записываемый в память компьютера

Длина строки

Длина строки — это всегда количество символов в строке, не включая нуль-символ.

Иногда вам может понадобиться выяснить длину строки. При определении длины строки нуль-символ никогда не берется в расчет. Даже притом, что нуль-символ должен завершать строку (чтобы компилятор C знал, где заканчивается строка символов), нуль-символ не является частью показателя длины строки.

Приняв во внимание определение строки, можно сказать, что все нижеприведенные строки имеют длину девять символов:

```
Wednesday
2 августа
I am here
```

При подсчете длины строки помните, что надо также посчитать все пробелы. Поэтому, хотя во второй строке общее количество цифр и букв равно восьми, а в третьей строке — семи, нужно дополнительно учесть один пробел посередине второй строки и два пробела в третьей строке. Так получается, что длина всех трех строк равна девяти символам. Если вы поместите три пробела между числом `2` и словом `августа` во второй строке, то получите 11-символную строку.

> **СОВЕТ**
>
> Длина единичных символьных данных равна 1. Поэтому длина строк 'X' и "X" равна 1, но строка "X" занимает в памяти два знакоместа из-за завершающего нуль-символа. Каждый раз, когда вы встречаете строковую константу, заключенную в кавычки (как и нужно делать по правилам), представляйте себе завершающий нуль-символ на конце строки, записываемый в память компьютера.

Символьные массивы: перечисление символов

Символьные массивы позволяют хранить в памяти компьютера строки текста. *Массив* — это особая разновидность переменной, о которой мы еще поговорим в следующих главах. У всех типов данных — `int`, `float`, `char` и пр. — есть соответствующий тип массивов. Массив, по сути — перечень переменных одного и того же типа.

Прежде чем использовать массив для сохранения строки текста, вы должны указать компилятору, что вам необходим символьный массив, причем сделать это нужно в том же участке кода, в котором указали бы необходимость использования переменной любого вида. После имени массива воспользуйтесь квадратными скобками (`[` и `]`), а также укажите число, соответствующее максимальному количеству символов, которое может содержать массив.

Впрочем, лучше один раз увидеть, чем сто раз услышать. Например, если бы вам было необходимо место для хранения названия месяцев, то вы могли бы объявить массив month следующим образом:

```
char month[10]; /* Объявление символьного массива */
```

СОВЕТ

В определении массивов нет ничего сложного. Если убрать квадратные скобки и число `10`, то у вас останется обычная символьная переменная. Добавление квадратных скобок и числа `10` сообщает компилятору языка C, что вам нужно 10 символьных переменных, следующих друг за другом по списку, название которого звучит как `month`.

При объявлении массива авторы использовали число `10` по той причине, что самое длинное название месяца в английском языке (`September`) содержит 9 символов. Десятый же символ нужен, вы уже, наверное, догадались, для нуль-символа.

СОВЕТ

Вы *всегда* должны резервировать достаточное количество места для хранения самой длинной строки из тех, что вам потенциально потребуется сохранить в массиве, плюс символ завершения строки.

Вы можете определить большее количество символов массива, чем нужно, но никогда не меньше.

При желании вы можете сохранить строковое значение в массиве сразу в момент определения этого массива, например:

```
char month[10] = "January"; /*Определение символьного массива*/
```

На рис. 6.2 представлен внешний вид этого массива. Так как в последние два знакоместа массива ничего не было записано (слово `January` занимает только семь знакомест плюс нуль-символ, занимающий восьмое место), вы не можете знать, что записано в последних двух участках памяти. (Однако некоторые компиляторы заполняют неиспользованные элементы массива нулями, чтобы отличить *пустые* элементы от остальной строки.)

Рис. 6.2. Определение и инициализация массива month, содержащего строковые данные

Каждый отдельный участок массива называется *элементом*. Массив `month` состоит из 10 элементов. Элементы можно различить по *индексу*. Индекс — это число, указываемое в квадратных скобках, ссылающееся на каждый отдельный элемент массива.

Нумерация элементов массива всегда начинается с 0. Как показано на рис. 6.2, первый элемент массива month называется `month[0]`, а последний — `month[9]`, так как общее количество элементов 10 и при отсчете с 0, последний, десятый, элемент массива будет иметь индекс `9`.

Все элементы символьного массива являются символами. Комбинация символов — массив, или *перечисление*, символов — содержит всю строку текста. При желании вы могли бы заменить содержание массива с `January` на `March`, меняя по очереди нужные элементы массива:

```
month[0] = 'M';
month[1] = 'a';
month[2] = 'r';
month[3] = 'c';
month[4] = 'h';
month[5] = '\0'; //Этот символ указывать обязательно
```

Вставка нуль-символа в конец строки является жизненно необходимой. Если этого не сделать, то массив month будет по-прежнему содержать нуль-символ на три символа дальше, в элементе `month[7]`. При попытке распечатать содержимое массива вы увидите на экране следующее:

```
Marchry
```

СОВЕТ

Вывод на печать строк из массивов также не представляет особой сложности, вам нужно лишь воспользоваться символом преобразования %s:

```
printf("Сейчас месяц: %s", month);
```

При *одновременном* объявлении *и* инициализации массива указывать число в квадратных скобках необязательно. Обе строки ниже абсолютно равнозначны:

```
char[8] = "January";
char[] = "January";
```

Во втором примере компилятор C автоматически подсчитает количество символов строке `January` и добавит одно знакоместо в массив для нуль-символа. Однако в дальнейшем вы не сможете сохранить в этот массив строку, содержащую более 8 символов.

Если же вы хотите определить массив строковых символов, инициализировать его, но оставить свободное место для возможности записать в массив более длинную строку, вы можете поступить следующим образом:

```
char month[25] = "January"; /* Резервируется место для
более длинных строк */
```

Инициализация строк

Конечно, инициализировать строку посимвольно, как было сделано в предыдущем разделе, не очень удобно. Однако, в отличие от обычных переменных, вы не можете записать новую строку в массив следующим образом:

```
month = "April"; /* так делать НЕЛЬЗЯ */
```

Записать строку в массив, воспользовавшись знаком равенства, можно *только в момент объявления массива*. Если далее по ходу программы вам понадобится записать новую строку в массив, то вам придется сделать это либо посимвольно, либо воспользоваться функцией языка C `strcpy()` (*string copy — копирование строки*), входящей в комплект поставки вашего компилятора. Следующее выражение записывает в массив `month` новую строку:

```
strcpy(month, "April"); /*Запись новой строки в массив*/
```

ПРИМЕЧАНИЕ

В тех программах, в которых вы планируете использование функции `strcpy()`, вам потребуется поместить следующую строку после строки `#include <stdio.h>`:

```
#include <string.h>
```

СОВЕТ

Не волнуйтесь, функция `strcpy()` автоматически добавит нуль-символ в конец создаваемой строки.

Теперь давайте соединим все то, что мы обсудили в этой главе, и создадим полноценную программу. Опять настало время как следует прогреть ваш редактор! Введите код, скомпилируйте его и запустите получившуюся программу:

```
// Программа-пример №1 из главы 6
// Руководства для начинающих по C, 3-е издание
// Файл Chapter6ex1.c
// Эта программа соотносит детей с их любимыми
супергероями
```

```
#include <stdio.h>
#include <string.h>
main()
{
char Kid1[12];
// Kid1 может содержать имя из 11 символов
// Kid2 будет содержать 7 символов (Maddie плюс нуль-
символ)
char Kid2[] = "Maddie";
// Kid3 также содержит 7 символов, но это указано
дополнительно
char Kid3[7] = "Andrew";
// Hero1 будет содержат 7 символов (учитывая
нуль-символ!)
char Hero1 = "Batman";
// Hero2 будет на всякий случай иметь дополнительное
пространство
char Hero2[34] = "Spiderman";
char Hero3[25];
Kid1[0] = 'K'; //Kid1 определяется посимвольно
Kid1[1] = 'a'; //Неэффективно, но работает
Kid1[2] = 't';
Kid1[3] = 'i';
Kid1[4] = 'e';
Kid1[5] = '\0'; // Не забывайте нуль-символ, чтобы
C знал, когда
// завершается строка
strcpy(Hero3, "The Incredible Hulk");
printf("%s\'s favorite hero is %s.\n", Kid1, Hero1);
printf("%s\'s favorite hero is %s.\n", Kid2, Hero2);
printf("%s\'s favorite hero is %s.\n", Kid3, Hero3);
return 0;
}
```

Как и в случае с программой в конце главы 5, вы можете задаться вопросом, зачем испытывать столько трудностей, определяя столько переменных, если можно просто-напросто вставить имена и строки текстов напрямую в функцию `printf()`?

Повторимся, ценность переменных будет более понятна после изучения главы 8, в которой вы научитесь получать информацию от пользователя.

Ранее вы уже использовали директиву `#include` для добавления файла `<stdio.h>` в программы, в которых применяется функция `printf()` (а также еще несколько функций, которые вы в скором времени добавите в свою коллекцию).

Теперь вы знаете еще один заголовочный файл, `<string.h>`, который вы также можете включать в свои программы. В следующей главе мы более детально обсудим директиву `#include`.

Чтобы напомнить вам о различных методах объявления и инициализации строковых переменных, мы определили переменные `kid` и `hero` различными способами.

Для развлечения закомментируйте строку кода с функцией `strcpy`, чтобы посмотреть, что ваша программа выведет на экран, когда массив `Hero3` передается в функцию `printf()` без инициализации. На нашем компьютере программа вывела на экран хаотический набор символов, которые уже были записаны в используемые массивом участки памяти, поэтому если вы сами ничего не запишете в массив, то получите на экране те символы, которые уже находятся в используемых им участках памяти.

Абсолютный минимум

Если вам нужно хранить слова в памяти компьютера, то вам придется воспользоваться символьными массивами. В языке C не поддерживается строковый тип данных. Ниже приведены основные моменты этой главы:

- Храните строки в символьных массивах, но при этом резервируйте достаточно места, чтобы иметь возможность сохранить потенциально самые длинные строки.

- Не забывайте, что строки должны заканчиваться завершающим нуль-символом.
- При записи данных в массив помните, что индексация начинается с 0, а не с 1.
- Существует три способа записи строки в массив: Инициализация массива одновременно с определением, присвоение значений элементам по одному, а также использование функции `strcpy()`.
- Если в своей программе вы используете функцию `strcpy()`, не забудьте добавить директиву `#include <string.h>` в начало программного кода.

Глава 7

#INCLUDE И #DEFINE: УВЕЛИЧЕНИЕ МОЩНОСТИ ВАШЕЙ ПРОГРАММЫ

В этой главе

- Подключение файлов к программе
- Использование директивы `#include`
- Определение констант
- Построение заголовочных файлов и программ

В программах на языке C можно часто видеть два типа инструкций, которые вовсе не являются командами. Они называются *директивами препроцессора*. Директивы препроцессора всегда начинается с символа «решетка» (#) и не изменяют ход выполнения программы (после того как она была запущена), вместо этого они работают во время компиляции программы.

Ниже приведены две наиболее часто используемые директивы препроцессора:

- `#include`
- `#define`

Во всех примерах программ, уже рассмотренных нами к настоящему моменту, содержалась директива `#include`. В этой главе мы наконец раскроем вам тайну секретной директивы препроцессора.

Включение файлов в программу

У директивы `#include` есть два практически идентичных формата:

```
#include <имя_файла>
```

и

```
#include "имя_файла"
```

На рис. 7.1 показана работа директивы `#include`. По сути эта директива является командой *слияния файлов*. Перед тем как будет завершена компиляция вашей программы, выражение `#include` заменится содержимым файла, указанным после `#include`. Если ваша операционная система позволяет, то имя файла в директиве может быть указано как заглавными, так и строчными буквами. Например, использованная авторами версия программы Code::Blocks для ОС Windows XP не видит различия между именами файлов, указываемыми строчными и прописными буквами, однако для систем UNIX это существенно. Если исходное имя файла выглядит как **myFile.txt**, то для его включения в программу вы можете указать любую из нижеследующих директив:

```
#include "MYFILE.TXT"
#include "myfile.txt"
#include "myFile.txt"
```

Однако при работе в операционной системе на платформе UNIX возможно указание только одного варианта директивы:

```
#include "myFile.txt"
```

Вот что вы написали:

Файл исходного кода:

```
    :
/* Часть программы на С */
age=31;
printf("Мне %d лет", age);
#include "addr.h";
printf("Это мой адрес");
/* Остальной код программы далее */
    :
```

Файл с именем addr.h:

```
printf("\n6104 E.Oak\n");
printf("St. Paul, MN\n");
printf("        54245\n");
```

ДО

Так видит вашу программу компилятор:

```
    :
/* Часть программы на С */
age=31;
printf("Мне %d лет", age);
printf("\n6104 E.Oak\n");
printf("St. Paul, MN\n");
printf("        54245\n");
printf("Это мой адрес");
/* Остальной код программы далее */
    :
```

ПОСЛЕ

Рис. 7.1. Директива `#include` вставляет содержимое одного файла с диска в другой файл

При установке компилятора программа-установщик устанавливает отдельное хранилище на жестком диске (в *папке* или *директории*) для

различных файлов #include, входящих в комплект поставки вашего компилятора. Если вы хотите воспользоваться одним из стандартных файлов #include, используйте формат директивы #include с угловыми скобками < и >.

ПРИМЕЧАНИЕ

Возможно, ранее при работе в текстовом процессоре вы уже использовали разновидность команд наподобие #include для вставки файла с диска в текст редактируемого файла.

ВНИМАНИЕ

Откуда мне знать, когда использовать стандартный файл #include? Хороший вопрос! Всем встроенным функциям, например printf(), соответствует определенный стандартный файл #include. При описании встроенных функций в этой книге мы также будем указывать, какой именно файл необходимо подключить (по-английски «include». — *Примеч. пер.*).

В своих программах вы уже использовали две встроенные функции: printf() и strcpy(). (Функция main() не является встроенной, ведь именно *вы* должны ее предоставить компилятору). Напомним, что стандартный файл #include для функции printf() — это **stdio.h** (что расшифровывается как *standard I/O*, или по-русски «стандартный ввод-вывод»), а для функции strcpy() — **string.h**.

Практически во всех полных листингах программного кода в этой книге содержится одна и та же директива препроцессора:

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

Так происходит по той причине, что почти во всех программах в этой книге используется функция printf(). В главе 6 мы уже говорили, что, когда бы вы ни использовали функцию strcpy(), вам необходимо подключать файл **string.h**.

СОВЕТ

Подключаемый вами файл называется *заголовочным файлом*. Именно поэтому имена почти всех подключаемых файлов имеют расширение .h (от английского слова «заголовок» — «header». — *Примеч. пер.*)

При написании собственных заголовочных файлов используйте вторую форму директивы препроцессора, с именем файла, заключенным в кавычки. При использовании кавычек компилятор С сначала ищет нужный файл в той папке, в которой сохранена ваша программа, и, если там не находит, продолжает поиск в директории `#include`, выбранной по умолчанию. Благодаря такому порядку поиска файлов вы можете создавать собственные заголовочные файлы и сохранять их с теми же именами, что и стандартные заголовочные файлы, так как в этом случае будет использован именно ваш файл, а не стандартный файл компилятора С.

ВНИМАНИЕ

При написании собственных заголовочных файлов не следует сохранять их в директорию для файлов `#include`, лучше вообще не трогать стандартные заголовочные файлы компилятора С. Очень редко появляется необходимость переопределения стандартных заголовков С, но вы можете добавлять собственные заголовочные файлы.

Если вы используете какое-то собственное выражение в большом количестве программ, вы также можете создать собственный заголовочный файл. Вместо того, чтобы вводить из с клавиатуры во всех создаваемых вами программах, вы можете просто включить их в текст программы, поместив соответствующие заголовочные файлы в папку с вашей программой и воспользовавшись директивой `#include` для автоматической вставки этих выражений в нужный вам файл.

Использование директивы *#include*

Заголовочные файлы, которые вы подключаете к своей программе с помощью директивы `#include`, по своей природе являются обычными текстовыми файлами, содержащими некий программный код на языке С. Чуть позже вы более подробно изучите содержимое заголовочных файлов, сейчас же важно понять, что заголовочный файл выполняет две вещи: во-первых, стандартные заголовочные файлы помогают компилятору С правильно выполнить встроенные функции, во-вторых, заголовочные файлы зачастую содержат написанный вами программный код, который вы можете использовать в нескольких программах.

СОВЕТ

Лучше помещать директивы `#include` перед функцией `main()`.

ПРИМЕЧАНИЕ

Покерная программа в приложении Б включает сразу несколько заголовочных файлов, так как в этой программе используется большое количество встроенных функций. Обратите внимание на то, в каком участке кода находятся директивы `#include`: все они употреблены перед функцией `main()`.

Определение констант

Для определения *констант* используется директива препроцессора `#define`. В языке C константа аналогична литералам. В главе 2 вы узнали, что литерал — это данные, не изменяющиеся по ходу выполнения программы, например число `4` или строка `"Программирование на C"`. Директива препроцессора `#define` позволяет присвоить литералам имена. Литерал, которому было присвоено имя, в терминологии программирования на языке C, называется *поименованной константой* или *определенной константой*.

ВНИМАНИЕ

В главе 5 вы узнали как определять переменные, указав их тип и присвоив им имя и начальное значение. Константы, определяемые с помощью директивы `#define`, *не являются переменными*, хотя иногда в процессе использования они очень похожи на переменные.

Ниже приведен формат директивы `#define`:

```
#define КОНСТАНТА определениеКонстанты
```

Как и в большинстве случаев, использование поименованных констант в языке C гораздо проще, чем может показаться на первый взгляд. Ниже приведены несколько примеров использования директивы `#define`:

```
#define AGELIMIT 21
#define MYNAME "Paula Holt"
#define PI 3.14159
```

В целом директива `#define` сообщает компилятору C примерно следующее: «Каждый раз, когда в программе появляется *`КОНСТАНТА`* заменить ее *`определениемКонстанты`*». Так, первая из приведенных выше директив #define сообщает компилятору заменить все вхождения слова `AGELIMIT` в программе на число `21`. Таким образом, если это выражение появится в программном коде после директивы `#define`, например:

```
if (employeeAge < AGELIMIT)
```

компилятор будет вести себя так, будто бы вы ввели

```
if (employeeAge < 21)
```

хотя вы этого не делали.

СОВЕТ

Для задания имен определенных констант используйте заглавные буквы. Пожалуй, это единственное исключение в языке C, когда желательно использовать заглавные буквы. Так как константы не являются переменными, использование заглавных букв позволяет вам быстро пробежаться глазами по программному коду и быстро определить, где переменная, а где константа.

Предположив, что ранее вы уже определили константу `PI`, использование заглавных букв в ее имени позволит вам избежать подобной ошибки в середине программного кода:

```
PI = 544.34; /* Запрещено */
```

Если имена всех определенных вами констант будут содержать только заглавные буквы, вы точно будете знать, что их значения менять нельзя, так как перед вами именно *константы*.

Использование поименованных констант очень удобно для поименования значений, которые могут изменяться между тестовыми запусками программы. Например, если бы вы не использовали поименованную константу `AGELIMIT` (возрастной предел. — *Примеч. пер.*), а вместо нее использовали бы число `21` в разных участках кода, то в случае изменения возрастного ограничения поиск и изменение каждого вхождения числа `21` было бы довольно трудоемким. Если же вы определили константу в самом начале кода и возрастной предел стало необходимо

изменить, то вам нужно будет лишь изменить выражение #define, например, следующим образом:

```
#define AGELIMIT 18
```

Директива #define не является командой языка С. Как и в случае с директивой #include, компилятор выполняет выражения #define до того, как компиляция программы будет завершена. Таким образом, если вы определили константу PI равной 3.14159 и использовали ее в нескольких участках кода, когда вам было необходимо значение математической константы пи (ϖ), компилятор будет считать, что вы ввели в этих участках кода именно число 3.14159, тогда как вы ввели лишь две буквы: PI. Имя константы PI гораздо проще запомнить (что позволяет уменьшить количество опечаток в коде), кроме того, такое имя делает более ясным предназначение константы.

В случае, если вы определили константу с помощью директивы #define перед функцией main(), то об этой константе будет знать вся программа. Таким образом, если вы определили константу PI равной 3.14159 до функции main(), то вы можете использовать константу PI на протяжении всего тела функции main() и любой другой создаваемой вами функции, следующей за функцией main(). При этом компилятор будет знать, что слово PI нужно заменять на число 3.14159 каждый раз перед компиляцией программы.

Построение заголовочных файлов и программ

Самый лучший способ убедиться в том, что вы поняли, что такое заголовочные файлы и поименованные константы, — это написать программу, в которой используется и то, и другое. Поэтому запустите редактор — и приступим!

Сначала создайте первый заголовочный файл:

```
//Программа-пример №1 из главы 7 Руководства
//по C для новичков, 3-е издание
//Файл Chapter7ex1.h
//Если у вас есть несколько значений, которые не будут
изменяться
//(или будут меняться редко),
```

```
//вы можете установить их с помощью директивы #DEFINE,
//что позволит изменять их по мере необходимости
//Если вы планируете использовать эти значения
//в нескольких программах, то вы можете
//поместить их в заголовочный файл
#define KIDS 3
#define FAMILY "Ивановы"
#define MORTGAGE_RATE 5.15
```

При редактировании и сохранении заголовочного файла укажите расширение **.h**, чтобы дать понять компилятору, что вы создали именно заголовочный файл, а не файл исходного кода программы. Большинство редакторов автоматически добавляют расширение **.c** к создаваемым вами файлам, если вы самостоятельно не указали иное.

Вы только что создали очень простой заголовочный файл с несколькими константами, устанавливаемыми директивой `#define`. Эти константы — отличный пример значений, которые вряд ли изменятся, но если все же перемена случится, то будет гораздо проще внести изменения в одном участке кода без необходимости редактировать сотни, если не тысячи, строк. Если вы создаете программы для планирования семьи, бюджета и праздничного шопинга и вдруг решили завести (или завели случайно) четвертого ребенка, то вы можете внести изменение в этот заголовочный файл, затем, когда вы перекомпилируете программы, использующие его, измененное значение `4` (или `5`, если вам повезло родить близнецов) пройдет по всему коду программ. Фамилия вряд ли изменится, но при рефинансировании ипотеки может измениться процентная ставка по ней, что повлияет на семейный бюджет и планирование налоговых выплат.

Чтобы задействовать заголовочный файл, его необходимо сначала подключить к программе. Ниже приведен простой образец кода, использующий только что созданный нами заголовочный файл **.h**:

```
//Программа-пример №1 из главы 7 Руководства
//по C для новичков, 3-е издание
//Файл Chapter7ex1.c
/* Это простая программа-пример, перечисляющая трех
детей, их потребности в школьных принадлежностях,
а также стоимость их приобретения */
```

```
#include <stdio.h>
#include <string.h>
#include "Chapter7ex1.h"
main()
{
  int age;
  char childname[14] = "Тимофей";

  printf("\n%s имеет %d детей", FAMILY, KIDS);

  age = 11;
  printf("Старший сын, %s, %d лет.\n", childname,
age);
  strcpy(childname, "Николай");
  age = 6;
  printf("Средний сын, %s, %d.\n", childname, age);
  age = 3;
  strcpy(childname, "Борис");
printf("Младший сын, %s, %d.\n", childname, age);
return 0;
}
```

Повторимся, этот код не выполняет практически ничего: программа лишь устанавливает, что в семье есть три ребенка, а затем выводит на экран имя каждого из них. Мы обещаем вам, что по мере изучения новых команд, выражений, функций и операторов в следующих главах ваши программы станут гораздо более увесистыми. Возможно, вы заметили, что одна из констант, определенных с помощью директивы `#define`, а именно `MORTGAGE_RATE`, не используется в программе. Запомните, что вам вовсе необязательно включать все имеющиеся в заголовочном файле константы в вашу программу.

В программе-примере используется одна переменная, `childname`, для сохранения имени и одна, `age`, — для сохранения возраста трех детей, при этом в каждом случае значения переменных перезаписываются. Однако это не самое лучшее решение, так как, если вы создаете программу, в которой используются имена ваших детей, скорее всего, вы будете

использовать эти имена не один раз. Но такая программа служит хорошим напоминанием, что вы можете перезаписывать и изменять значения переменных, но не констант, создаваемых с помощью директивы `#define`.

Абсолютный минимум

Директивы препроцессора C генерируют программный код на языке C, который вы на самом деле не создавали. Ниже приведены основные концепции этой главы.

- При использовании встроенных функций всегда подключайте к программе соответствующие заголовочные файлы с помощью директивы `#include`. Заключайте имя включаемого файла в угловые скобки (< и >) при подключении стандартного заголовочного файла, входящего в комплект поставки компилятора. Кроме того, убедитесь, что директивы `#include` для подключения таких файлов находятся перед функцией `main()`.
- При подключении собственных заголовочных файлов, которые вы сохранили в той же папке, что и программный код, заключайте имена таких файлов в кавычки (`"`). Собственные заголовочные файлы вы можете включать в любое место программного кода также с помощью директивы `#include`.
- Используйте только заглавные буквы для задания имен определенных констант, это позволит вам отличить их от переменных.

Глава 8
ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ

В этой главе

- Обзор функции `scanf()`
- Запрос ввода данных функции `scanf()`
- Решение проблем, связанных с функцией `scanf()`

Функция `printf()` выводит данные на экран, в то время как функция `scanf()` принимает данные, вводимые с клавиатуры. При необходимости у вас должна быть возможность организовать получение данных программой от пользователя, ведь вы не можете постоянно назначать значения переменных с помощью операций присваивания. Например, если бы вам поручили написать программу для кинотеатров, которую использовали бы по всей стране, вы не смогли бы установить стоимость билета с помощью операции присваивания, так как стоимость билетов в разных кинотеатрах может различаться. Вместо этого вам понадобилось бы запрашивать у пользователя программы (в нашем случае кинотеатров) стоимость билета перед расчетом окончательного тарифа.

На первый взгляд функция `scanf()` может показаться сложной, но она очень важна для увеличения мощности ваших программных продуктов через добавление интерактивности (общения с пользователем). Для начинающего программиста функция `scanf()` не очень важна, но, несмотря на странный формат (синтаксис), это на данном этапе вашего обучения самая простая функция для получения пользовательского ввода, так как она имеет очень тесные связи с функцией `printf()`. Практика работы с функцией `scanf()` сделает ваши программы по-настоящему совершенными!

Обзор функции *scanf()*

Функция `scanf()` — это встроенная функция языка С, входящая в комплект поставки всех компиляторов С. Для этой функции исполь-

зуется тот же заголовочный файл, что и для функции printf(), а именно stdio.h, поэтому вам не придется волноваться о включении дополнительного заголовочного файла специально для функции scanf(). Функция scanf() присваивает переменным данные, введенные пользователем.

Функция scanf() достаточно проста, если вы уже знаете функцию printf(). Внешний вид функции scanf() очень похож на то, как выглядит функция printf(), так как в обеих функциях используются коды преобразования, такие как %s и %d. Можно сказать, что функция scanf() является зеркальным отражением функции printf(). Очень часто вы будете писать программы, запрашивающие данные для вывода на экран с помощью функции printf() и получающие эти данные с использованием функции scanf(). Ниже приведен формат функции scanf():

```
scanf(контрольнаяСтрока [, данные]);
```

Когда ход выполнения программы достигает функции scanf(), выполнение программы приостанавливается, и компьютер ожидает, пока пользователь введет необходимые данные. Переменные, перечисленные в функции scanf() (сразу после *контрольнойСтроки*) получат любые введенные пользователем данные. Выполнение функции scanf() завершается, когда пользователь нажимает клавишу **Enter**.

Несмотря на то, что в функции scanf() используются те же символы преобразования, что и в функции printf(), никогда не используйте управляющие последовательности, такие как \n, \a или \t, так как управляющие последовательности только запутывают компилятор. Выполнение функции scanf() завершается при нажатии клавиши **Enter**, поэтому дополнительно указывать управляющую последовательность \n не нужно.

Запрос ввода данных функции *scanf()*

Почти каждый раз перед использованием функции scanf() вам потребуется использовать функцию printf(). Если этого не сделать, то программа прервется в ожидании ввода данных от пользователя, при этом пользователь не будет знать, что делать дальше. Например, если вам требуется получить от пользователя некое количество, то перед использованием функции scanf() вы должны будете употребить функцию printf() примерно следующего содержания:

```
printf("Какое количество необходимо? "); /* Запрос */
/* Функция scanf() далее */
```

Функция `printf()`, употребленная непосредственно перед `scanf()`, отправляет на экран пользователя некий запрос. Если вы не запросите у пользователя нужное значение или группу значений, то пользователь не будет знать, что именно он должен ввести в программу. В общем, можно сказать, что функция `printf()` запрашивает данные от пользователя, а `scanf()` — принимает данные, введенные пользователем в ответ на запрос.

Давайте напишем простую программу с несколькими простыми выражениями `scanf()`, ведь, в конце концов, практика — лучший способ чему-либо обучиться:

```
// Программа-пример №1 из главы 8 «Руководства по C
// для новичков», 3-е издание
// Файл Chapter8ex1.c
/* Эта программа-пример запрашивает у пользователей
некоторые основные данные, после чего выводит их на
экран, чтобы показать, что было введено */
#include <stdio.h>
main()
{
  // Установка переменных, значения которым будут
//присвоены функцией scanf()
char firstInitial;
char lastInitial;
int age;
int favorite_number;
printf("С какой буквы начинается Ваша фамилия?\n");
scanf(" %c", &firstInitial);
printf("С какой буквы начинается Ваше имя?\n");
scanf(" %c", &lastInitial);
printf("Сколько Вам лет?\n");
scanf(" %d", &age);
printf("Какое Ваше любимое число? (Только целое)\n");
```

```
scanf(" %d", &favorite_number);
printf("\n\nВаши инициалы: %c.%c., и вам %d лет",
firstInitial, lastInitial, age);
printf("\nВаше любимое число %d. \n\n", favorite_
number);
return 0;
}
```

Выходит, выражения с функцией `scanf()` не так уж и плохи, да? У каждого из таких выражений есть выражение-партнер с функцией `printf()`, говорящее пользователю, какую именно информацию необходимо ввести. Для того чтобы понять, насколько запутанными могут быть выражения `scanf()` без предшествующего выражения `printf()`, закомментируйте одно из выражений `printf()`, следующих перед функцией `scanf()`, перекомпилируйте и запустите программу. Программа стала непонятной, запросы сбивают с толка, а ведь это вы ее написали! Представьте, как чувствовал бы на вашем месте пользователь такой программы.

Первые два выражения `scanf()` принимают символьные значения (это можно понять благодаря кодам преобразования `%c`). Третье выражение `scanf()` принимает вводимое с клавиатуры целочисленное значение и записывает его в переменную с именем `age`.

В переменные `firstInitial`, `lastInitial` и `age` будут записаны любые значения, введенные пользователем с клавиатуры перед нажатием клавиши **Enter**. Если пользователь введет в первые два запроса два и более символов, то это может сбить с толку программу и создать проблемы для следующих вводимых значений.

Еще один важный нюанс о выражениях с функциями `scanf()` — это пробел перед каждым символом `%c` и `%d`. Пробел не всегда обязателен, но он никогда не повредит, а иногда он даже улучшает функциональность ввода, особенно когда необходимо вводить сразу и числа и символы. Добавление дополнительного пробела — это хорошая привычка, к которой следует приучиться уже сейчас, во время первого знакомства с функцией `scanf()`.

Хватит об этом. Давайте перейдем к самой очевидной проблеме с функцией `scanf()` — символу амперсанда (&), употребляемому перед тремя переменными. Угадайте что? Функция `scanf()` требует, чтобы вы всегда употребляли амперсанд перед всеми переменными, даже несмотря на то, что амперсанд *не является* частью имени переменной.

Сделайте, как вас просят — и функция `scanf()` будет работать, но стоит забыть про амперсанд — и функция `scanf()` не примет пользовательский ввод и не запишет его в переменные.

> **СОВЕТ**
>
> Выражения `printf()` всегда следует делать настолько описательными, насколько это возможно. В предыдущем примере, если вы попросите ввести просто любимое число, то пользователь может ввести десятичную дробь вместо целого числа: кто знает, может, 3,14159 — чье-то любимое число.

Проблемы с функцией *scanf()*

Как уже отмечалось ранее в этой главе, функция `scanf()` — не самая простая в использовании. Одна из проблем с функцией `scanf()` заключается в том, что, хотя, по идее, пользователь должен ввести именно то, что от него требует функция `scanf()`, пользователи редко так поступают. Если функции `scanf()` требуется число с плавающей точкой, а пользователь вводит символ, вы мало что можете с этим сделать. В предоставленную вами переменную для записи числа с плавающей точкой будут записаны плохие данные, потому что символ — это не число с плавающей точкой.

Сейчас давайте сделаем допущение, что пользователь *действительно* вводит то, что от него требуется. В главе 18 описываются способы преодоления проблем с функцией `scanf()` (хотя многие современные программисты зачастую полагаются на комплексные процедуры ввода данных, создаваемые ими, загружаемые или приобретаемые у третьих источников, позволяющие преодолевать сложности языка C в области ввода данных).

К правилу амперсанда есть одно исключение. Если вы принимаете ввод в массив с помощью символа `%s`, например запросив у пользователя ввести имя, сохраняемое в символьном массиве, амперсанд использовать *не нужно*.

Проведем черту следующим образом: если вы просите пользователя ввести целые числа, дробные числа, символы, дробные числа удвоенной точности типа `double` или прочие комбинации, сохраняемые в одной переменной («длинные» целые числа типа `long` и т. д.), употребляйте амперсанд перед именами переменных при их использовании в функ-

ции `scanf()`. Если же вы просите пользователя ввести данные для сохранения в символьном массиве, то перед именем массива амперсанд употреблять не нужно.

ВНИМАНИЕ

Перед переменными-*указателями* вы также не будете употреблять амперсанд. На самом деле, массив по своей природе является переменной-указателем, именно поэтому для массива не требуется употребление амперсанда. К изучению указателей вы приступите чуть позже, но если вы встречались с ними в других языках программирования, то вы понимаете, о чем идет речь. Если вы никогда не встречались с переменными-указателями и не понимаете, что это такое, что ж, обещаем, скоро вы до этого дойдете! Вы полностью будете понимать, что такое указатели и чем они похожи на массивы, после прочтения главы 25.

Существует проблема использования функции `scanf()` для записи строк в массивы символов. Функция `scanf()` прекращает чтение строкового ввода после первого пробела, поэтому при помощи `scanf()` за один раз в массив можно ввести только одно слово. Если вам нужно запросить у пользователя больше одного слова, например его имя и фамилию, то вам нужно воспользоваться двумя выражениями `scanf()` (каждому из которых должно предшествовать выражение `printf()` с пояснением запроса), а также сохранить имя и фамилию в двух отдельных массивах.

В следующей программе используются выражения `scanf()` для запроса ввода числа с плавающей точкой (цены пиццы), строки (желаемая начинка пиццы), а также нескольких целых чисел (количество кусков пиццы, а также месяц, день и год). Обратите внимание, что перед строковой переменной не употребляется амперсанд, указываемый перед именами всех остальных переменных. Программа запрашивает однословное описание начинки пиццы, так как функция `scanf()` не может принять два слова за раз.

```
// Программа-пример №2 из главы 8 Руководства по C
// для новичков, 3-е издание
// Файл Chapter8ex2.c
/* Эта программа-пример запрашивает у пользователей
некоторые основные данные, после чего выводит их на
экран, чтобы показать, что было введено */
```

```
#include <stdio.h>
main()
{
  char topping[24];
  int slices;
  int month, day, year;
  float price;
// Первая scanf ищет переменную
// типа float для сохранения
// стоимости пиццы
// Если пользователь не введет
// знак $ перед ценой,
// возникнут проблемы
  printf("Сколько стоит пицца в вашем районе?");
  printf("\n(Введите как $XX.XX)\n");
  scanf(" $%f", &price);
//Начинка пиццы – строка, поэтому & не нужен
  printf("Какая ваша любимая начинка (одним
словом)?\n");
  scanf(" %s", topping);
  printf("Сколько кусков пиццы с начинкой %s",
topping);
  printf("вы можете съесть за раз?\n");
  scanf(" %d", slices);
  printf("Какое сегодня число? (Введите в формате XX/
XX/XX) \n);
  scanf(" %d/%d/%d", &month, &day, &year);
  printf("\n\nПочему бы не угостить себя ужином
%d/%d/%d", month, day, year);
  printf("\nи не съесть %d кусков пиццы с начинкой
%s?\n", kuski, nachinka);
  printf("Это будет стоить лишь $%.2f!\n\n\n", cena);
  return (0);
}
```

Формат и использование функции `scanf()` станут для вас проще с практикой. Если бы пользователь захотел ввести описание начинки, состоящее из двух слов, например «Итальянская колбаса», то вашей программе для обработки этого ввода и сохранения потребовалось бы два выражения `scanf()` и две переменные. Чуть позже в этой книге мы познакомим вас с несколькими трюками запроса нескольких элементов данных, а не одного.

Повторимся, используйте функцию `printf()`, чтобы более точно направлять пользователя и указать ему, в каком формате требуется ввести данные в программу. Попробуйте, запустив программу, ввести данные в неверном формате, например забыв употребить знак доллара при указании цены пиццы или слеши при задании даты, — и вы увидите, какие проблемы в выполнении программы это повлечет.

СОВЕТ

Вы можете позволить пользователям вводить дополнительные символы, не являющиеся данными. Например, очень часто даты вводятся с косыми чертами (слешами) или тире для разделения порядковых номеров дней, месяцев и лет, например так: 05/03/95. Вам придется довериться пользователю, что он введет все данные так, как нужно. В предыдущем примере, если пользователь не введет знак доллара перед стоимостью пиццы, то программа будет функционировать неправильно. Следующее выражение `scanf()` ожидает, что пользователь введет дату именно в формате *дд/мм/гг*:

```
scanf(" %d/%d/%d", &day, &month, &year);
```

Пользователь мог бы ввести дату `28/02/14` или `22/11/13`, но не `5-е июня 2013`, т.к. функция `scanf()` ожидает другого.

Абсолютный минимум

Цель этой главы научить вас, как задавать запросы и получать на них ответы пользователя. Возможность обработки пользовательского ввода — это очень важная часть любого языка программирования. Функция `scanf()` осуществляет ввод данных в программу, т.е. функция `scanf()` принимает пользовательский ввод и сохраняет этот ввод в переменные. Ниже приведены основные моменты повествования этой главы:

- Используйте функцию `scanf()` для получения данных от пользователя путем их ввода с клавиатуры, не забудьте использовать контрольную строку для обозначения того, как будут выглядеть вводимые данные.
- Прежде чем использовать функцию `scanf()`, воспользуйтесь функцией `printf()`, чтобы запросить у пользователя ввод данных в нужном вам формате.
- В функции `scanf()` употребляйте амперсанд (&) перед именами переменных, но не массивов.
- Всегда оставляйте пробел перед первым символом контрольной строки (как пример строка `"  %d"`, содержащая пробел перед `%d`) для обеспечения корректного ввода символов.

Глава 9
ЧИСЛА: МАТЕМАТИЧЕСКИЕ ОПЕРАЦИИ В С

В этой главе

- Основные арифметические операции
- Порядок выполнения математических операций
- Нарушение правил с помощью скобок
- Повсеместное использование присваивания

Очень многих людей до сих пор бросает в дрожь, когда они слышат о необходимости произведения каких-либо математических вычислений. К счастью, компьютеры совсем не против вычислений, все, что вам нужно — это вводить правильные числа, а ваша программа на С всегда выполнит верный расчет с помощью различных операторов. Термин *оператор* может вызывать ассоциацию «барышень», которые когда-то помогали вам совершать междугородние и международные телефонные звонки, но мы будем обсуждать не их. Мы будем говорить об операторах языка С, помогающих вам совершать математические операции.

Вам нужно не только научиться распознавать сами математические операторы языка С, но также вы должны будете выучить порядок выполнения математических операций в С. В языке С вычисления не всегда производятся слева направо. В этой главе мы объясним вам, почему.

Базовые арифметические операции

Большое количество операторов языка С работают именно так, как вы ожидаете, чтобы они работали. Если вам нужно сложить числа, воспользуйтесь знаком плюс (+), воспользуйтесь знаком минус (-), чтобы выполнить вычитание. Программисты, работающие на языке С, часто используют математические выражения в правой части оператора присваивания для записи значений в переменные, например вот так:

```
totalSales = localSales + internationalSales –
salesReturs;
```

Компилятор С вычисляет ответ и сохраняет этот ответ в переменную `totalSales`.

ПРИМЕЧАНИЕ

Если вы хотите вычесть отрицательное значение, не забудьте разделить знаки минус пробелом:

```
newValue = oldValue - -factor;
```

Если опустить этот ответ, то компилятор сочтет, что вы используете другой оператор, `--`, который называется оператором *декремента*, описываемый в главе 13.

Математическое выражение можно даже поместить в функцию `printf()`:

```
printf("Через 3 года мне будет %d лет.\n", age+3);
```

Для умножения и деления используются символы `*` и `/` соответственно. Следующее выражение присваивает значение переменной, используя операции умножения и деления:

```
newFactor = fact * 1.2 / 0.5;
```

ВНИМАНИЕ

Если по *обеим* сторонам оператора деления (`/`) поместить целые числа, то компилятор С вычислит *целочисленное частное от деления*. Изучите следующую программу, чтобы познакомиться с целочисленным делением и обычным делением. Комментарии поясняют результаты, полученные после выполнения деления, но вы всегда можете дополнительно удостовериться, самостоятельно скомпилировав и запустив программу на компьютере.

```
//  Программа-пример №1 из главы 9
//  Руководства по С для новичков, 3-е издание
//  Файл Chapter9ex1.c
/* Эта программа-пример демонстрирует математические
операторы, а также различные виды деления. */
```

```
#include <stdio.h>
main()
{
  // Два набора одинаковых переменных, один набор
  // целочисленный, второй - с плавающей точкой
  float a = 19.0;
  float b = 5.0;
  float floatAnswer;
  int x = 19;
  int y = 5;
  int intAnswer;
  // Использование двух переменных типа float дает
ответ 3.8
  floatAnswer = a / b;
printf("%.1f разделить на %.1f получится %.1f\n", a,
b, floatAnswer);
  floatAnswer = x / y; //Вторая проба дает ответ 3.0
printf("%d разделить на %d получится %.1\n", x, y,
floatOtvet);
  // Это выражение тоже даст ответ 3, но это лишь
// отсечение дробной части, но не математическое
округление
intAnswer = a / b;
printf("%.1f разделить на %.1f получится %.1f\n", a,
b, intAnswer);
intAnswer = x%y; //Это выражение вычисляет остаток от
деления (4)
printf("Остаток от деления %d на %d равняется %d", x,
y, intAnswer);
return 0;
}
```

Возможно, последнее математическое выражение в программе покажется вам новым. Если вам необходимо узнать остаток от деления, то вы можете воспользоваться оператором языка C *деление по модулю* (`%`). Воспользовавшись значениями переменных из предыдущего листинга,

можно сделать вывод, что следующее выражение присваивает переменной `ansMod` значение 4:

```
ansMod = x % y; /* 4 - остаток от деления 19 / 5 */
```

Теперь вы знаете, что в языке С предусмотрено три способа деления чисел: обычное деление (если хотя бы с одной стороны оператора / находится число с плавающей точкой), целочисленное деление (если по обеим сторонам оператора / только целые числа), а также деление по модулю, если между двумя целыми числами используется оператор %.

СОВЕТ

Оператор % не используется с нецелочисленными переменными

Следующая небольшая программа вычисляет чистую выручку от продажи автомобильных покрышек:

```
//  Программа-пример №2 из главы 9
//  Руководства по С для новичков, 3-е издание
//  Файл Chapter9ex2.c
/* Эта программа запрашивает у пользователя количество
приобретенных покрышек и стоимость одной покрышки,
после чего программа вычисляет чистую выручку,
прибавив к ней налог с продаж (sales tax). */
// Если вы обнаружите, что ставка налога с продаж
изменилась,
// воспользуйтесь директивой #define,
// чтобы изменить ее только в одном месте
#include <stdio.h>
#define SALESTAX .07
main()
{
  // Переменные для хранения количества проданных
покрышек,
  // цены, подитога без учета налогов и общей
стоимости
  // с учетом налога
  int numTires;
```

```
  float tirePrice, beforeTax, netSales;
  /* Получить количество купленных покрышек
и стоимость покрышки */
  printf("Сколько покрышек вы приобрели? ");
  scanf(" %d", &numTires);
  printf("Какова стоимость одной покрышки? (В формате
$XX.XX)? ");
  scanf(" $%f", &tirePrice);
  /* Подсчет цены */
  beforeTax = tirePrice + numTires;
  netSales = beforeTax + (beforeTax * SALESTAX);
  printf("На покрышки вы потратили $%.2f\n\n\n",
netSales);
  return 0;
}
```

Ниже приведен результат тестового запуска программы:

```
Сколько покрышек вы приобрели? 4
Какова стоимость одной покрышки? (В формате $XX.XX)?
$84.99
На покрышки вы потратили $363.76
```

Порядок выполнения математических операций

Как мы уже упоминали ранее в этой главе, математические операции в языке С не всегда выполняются строго в том порядке, в котором вы ожидаете, что они будут выполнены. Следующее выражение очень доходчиво все объясняет:

```
ans = 5 + 2 * 3;  /* Записывает 11 в переменную ans*/
```

Если вы подумали, что С запишет в переменную `ans` значение `21`, значит, вы прочитали это выражение слева направо. Однако в языке С умножение всегда выполняется перед сложением. Звучит дико, но если знать эти правила, то все будет хорошо. Компилятор С всегда следует таблице *порядка выполнения операций*. Сначала компилятор умножит 2 на 3, получив 6, после чего прибавит 5 к 6 — и получит ответ 11.

В табл. 9.1 приведен порядок выполнения всех операций. (В таблицу включены несколько операторов, которые мы еще не изучали, впрочем, не стоит переживать по этому поводу, по ходу чтения книги вы изучите их все.) На каждом уровне, если ваше выражение содержит несколько операторов одного уровня, С вычисляет значение выражения, используя приведенное далее ассоциативное направление. Таким образом, если вам необходимо произвести умножение и деление, то С выполнит первой ту операцию, которая следует первой при чтении выражения слева направо.

После завершения всех операций текущего уровня компьютер переходит к выполнению операций нижеследующего уровня. Как вы можете видеть в таблице, операторы `*`, `/` и `%` приведены перед операторами + и `-`.

Табл. 9.1. Порядок выполнения операций

Уровень	**Оператор**	**Ассоциативность**
1	`()` (скобки), `[]` (элемент массива), `.` ссылка на член структуры	Слева направо
2	`-` (знак отрицательного числа), `++` (инкремент), `--` (декремент), `&` (взятие адреса), `*` (разыменование указателя), `sizeof()`, `!` (логическое НЕ)	Справа налево
3	`*` (умножение), `/` (деление), `%` (деление по модулю)	Слева направо
4	`+` (сложение), `-` (вычитание)	Слева направо
5	`<` (меньше), `<=` (меньше или равно), `>` (больше), `>=` (больше или равно)	Слева направо
6	`==` (равно), `!=` (неравно)	Слева направо
7	`&&` (логическое И)	Слева направо
8	`||` (логическое ИЛИ)	Слева направо
9	`?:` (оператор условия)	Справа налево
10	`=`, `*=`, `/=`, `%=`, `+=`, `-=` (операторы присваивания)	Справа налево
11	`,` (оператор-запятая)	Слева направо

Ниже приведено более сложное выражение. Все переменные и числа целые. Посмотрим, сможете ли вы понять, сможете ли вы его решить так, как решила бы его С:

```
ans = 5 + 2 * 4 / 2 % 3 + 10 - 3; /* Каков ответ? */
```

На рис. 9.1 показано, как решить это выражение, чтобы получить ответ `13`.

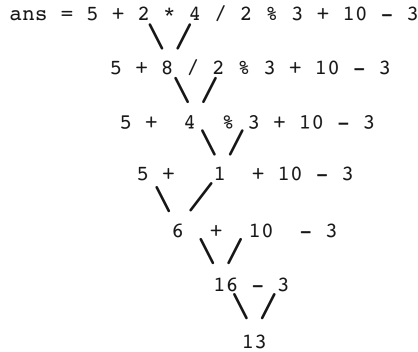


Рис. 9.1. Решение выражения по правилам языка C

СОВЕТ

Когда будете практиковаться в решении подобных уравнений, не стоит выполнять большое количество действий сразу. Как показано на рис. 9.1, необходимо выполнять только одно действие за раз, после чего переносить результаты этого действия в выражение для следующего раунда вычислений.

Если выражение, подобное тому, что было приведено на рис. 9.1, содержит сразу несколько операторов одного уровня, то вам необходимо пользоваться третьей колонкой таблицы (Ассоциативность) для установления того, в каком порядке следует выполнять операции. Иными словами, поскольку операторы `*`, `/` и `%` находятся на одном уровне, то они должны выполняться слева направо, как сказано в колонке «Ассоциативность».

Возможно, вы зададитесь вопросом, зачем вам изучать все это, в конце концов, разве C не должен вычислять всю математику за вас? Наш ответ: «Да, *но*...» Конечно, программа, написанная на C, выполнит всю математику, но вы должны знать, как правильно составить математическое выражение. Классический довод звучит следующим образом: допустим, вам нужно подсчитать среднее трех чисел, следующее выражение *не даст* правильного ответа:

```
avg = i + j + k + 1 / 4;  /* среднее посчитано НЕ
будет */
```

Причину найти просто, если вы знаете порядок выполнения операций. В языке C сначала выполняется деление, то есть первым выполня-

ется деление `1 / 4`, только затем складываются переменные `i`, `j` и `k` и их сумма прибавляется к частному от деления. Если вы хотите переназначить порядок выполнения операций, то вам нужно выучить правила расстановки скобок в выражениях.

Нарушение правил с помощью скобок

При необходимости порядок выполнения математических операций можно изменить. Как показано в табл. 9.1, если несколько операторов заключить в скобки, то С сначала выполнит операторы в скобках, а потом все остальные. Так как в таблице порядка выполнения математических операций скобки приводятся перед всеми остальными операторами, у скобок наивысший приоритет, как показано на следующем примере:

```
ans = (5 + 2) * 3; /* Присваивает переменной ans
значение 21 */
```

Даже несмотря на то, что обычно умножение выполняется перед сложением, скобки заставляют программу сначала подсчитать сумму `5 + 2`, после чего умножить получившуюся сумму `7` на `3`. Таким образом, если вам необходимо усреднить четыре значения, то вы должны сначала сгруппировать все операторы сложения и заключить их в скобки:

```
avg = (i + j +k + 1) / 4; /* Подсчет среднего значения */
```

СОВЕТ

Рекомендуем использовать большое количество скобок: они позволяют сделать ваши математические выражения более понятными. Даже если для вычисления правильного ответа будет достаточно обычного порядка выполнения математических операций, использование скобок поможет вам расшифровать ваши выражения в дальнейшем, если возникнет необходимость в изменении программного кода.

Повсеместное использование присваивания

Как можно видеть из таблицы, оператору присваивания также характерна ассоциативность и определенная приоритетность, как и всем остальным операторам. В таблице у оператора присваивания очень низкий приоритет, а ассоциативность справа налево.

Ассоциативность справа налево позволяет вам выполнить интересный трюк: в одном выражение вы можете присвоить значение сразу нескольким переменным. Так, чтобы присвоить значение 9 десяти различным переменным, вы могли бы поступить следующим образом:

```
a = 9; b = 9; c = 9; d = 9; e = 9; f = 9; g = 9; h =
9; i = 9; j = 9;
```

но гораздо проще сделать вот так:

```
a = b = c = d = e = f = g = h= i = j = 9;
```

Благодаря направлению ассоциативности справа налево, сначала значение 9 присваивается переменной j, затем переменной i и так далее.

ПРИМЕЧАНИЕ

В языке С переменные не инициализируются автоматически, если вам нужно присвоить значение 0 определенному набору переменных, вы можете осуществить это с помощью операции множественного присваивания.

Результатом всех выражений С всегда являются значения. Выражение j = 9; присваивает переменной j значение 9, но, кроме того, результатом выполнения этого выражения является полноценное завершенное значение 9, которое также можно пересохранить в любом другом месте, если необходимо. Тот факт, что результатом операции присваивания является выражение, позволяет вам выполнять трюки, наподобие приведенного ниже, которые не всегда доступны в других языках программирования:

```
a = 5 * (b = 2); /* Записывает 2 в b, а затем 10 в a
*/
```

Следующая программа — это еще один (последний) пример использования операции присваивания, математических операторов и скобок, изменяющих порядок выполнения операций:

```
//  Программа-пример №3 из главы 9
//  Руководства по С для новичков, 3-е издание
//  Файл Chapter9ex3.c
```

```
/* Эта программа вычисляет среднюю из четырех оценку,
а также выполняет другие базовые математические
операции */
#include <stdio.h>
main()
{
int grade1, grade2, grade3, grade4;
float averageGeade, gradeDelta, percentDiff;
/* Студент получил 88 баллов за первый и третий тесты,
поэтому подойдет выражение множественного присваивания
*/
grade1 = grade3 = 88;
grade2 = 79;
//Пользователь должен ввести четвертую оценку
printf("Сколько Вы получили за четвертый тест");
printf(" (Целое число в диапазоне от 0 до 100)?");
scanf(" %d", &grade4);
averageGrade = (grade1+grade2+grade3+grade4)/4;
printf("Ваша средняя оценка %.1f\.n", averageGrade);
gradeDelta = 95 - averageGrade;
percentDiff = 100 * ((95-averageGrade) / 95);
printf("Ваша оценка на %.1f баллов ниже, чем ",
gradeDelta);
printf(" - лучшая оценка в классе\n");
printf("Ваш результат на %.1f процентов хуже, ",
percentDiff);
printf("чем эта оценка!\n\n\n");
return 0;
}
```

Эта программа позволяет закрепить использование операторов присваивания, а также операторов сложения, вычитания, умножения и деления. В этой программе вы также воспользовались скобками для установки собственного порядка выполнения математических операций, в том числе и двойные скобки при вычислении процентной разницы между средней оценкой пользователя и самой высокой оценкой в клас-

се. Продолжайте практиковаться с этими программами на C — и у вас будет лучшая оценка за программирование в классе!

Абсолютный минимум

В языке C представлены несколько математических операторов, выполняющих за вас математические вычисления. Вам нужно лишь понимать порядок выполнения математических операций с тем, чтобы знать наверняка, что для выполнения нужных расчетов вы верно ввели исходные данные (значения). Ниже перечислены основные моменты этой главы.

- Для сложения, вычитания, умножения и деления используйте соответственно операторы `+`, `-`, `*` и `/`.
- Воспользуйтесь оператором «деление по модулю» (`%`) для получения остатка от деления целых чисел.
- Помните о порядке выполнения операций, пользуйтесь скобками для изменения порядка их выполнения.
- При вычитании отрицательного числа не забудьте разделить два минуса пробелом, иначе компилятор примет их за другой оператор.
- Если вам нужно инициализировать несколько переменных, воспользуйтесь оператором множественного присваивания.

Глава 10

ПРИДАНИЕ СИЛЫ ПЕРЕМЕННЫМ С ПОМОЩЬЮ ПРИСВАИВАНИЙ И ВЫРАЖЕНИЙ

В этой главе

- Составные операторы и экономия времени
- Составные операторы и порядок выполнения операторов
- Приведение типов переменных

Как вы можете видеть из табл. 9.1 предыдущей главы, в языке C предусмотрен богатый ассортимент операторов. Благодаря многим из них в языке C удается сохранять сравнительно небольшим набор командных слов. На самом деле в языке C нет большого количества команд, в этом языке гораздо больше операторов, чем в большинстве языков программирования, в то время как в большинстве языков программирования сравнительно мало операторов и очень много команд. Благодаря большому количеству предлагаемых операторов языку C удается сохранять свою минималистскую натуру.

Составной оператор присваивания

Очень часто в программах вам придется изменять значения переменных. До этого момента мы присваивали переменным значения литеральных констант или выражений, однако достаточно часто у вас будет возникать необходимость в обновлении значения переменной.

Предположим, что ваша программа должна подсчитать количество раз, когда уровень дохода находился ниже нулевой отметки. Для этого вам понадобилась бы *переменная-счетчик*. Счетчик — это особая переменная, к которой прибавляется 1 всякий раз, когда наступает определенное событие. Каждый раз, когда значение дохода представлено отрицательным числом, вы можете выполнить следующее:

```
lossCount = lossCount + 1; /* Прибавляет 1 к значению
переменной lossCount */
```

ВНИМАНИЕ

В классической математике переменная не может равняться самой себе плюс 1. Однако в случае с информатикой и компьютерами предыдущее выражение сначала прибавляет единицу к уже имеющемуся значению переменной `lossCount`, после чего записывает получившийся результат в переменную `lossCount`, по сути, это эквивалентно простому прибавлению единицы к переменной `lossCount`. Запомните, что знак равно означает «взять правую часть выражения, вычислить значение и записать его в переменную в левой части».

Следующая простая программа выводит на экран числа от 1 до 5, используя при этом перед каждым вызовом функции `printf()` операцию присвоения со счетчиком, а затем выполняет обратный отсчет до 1:

```
//  Программа-пример №1 из главы 10
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter10ex1.c
/* Эта программа увеличивает значение счетчика от 1 до
5, при этом, выводя на печать обновленные значения,
после чего выполняет обратный отсчет до 1 */
#include <stdio.h>
main()
{
int ctr = 0;
ctr = ctr + 1; //увеличивает счетчик до 1
printf("Значение счетчика: %d. \n", ctr);
ctr = ctr + 1; //увеличивает счетчик до 2
printf("Значение счетчика: %d. \n", ctr);
ctr = ctr + 1; //увеличивает счетчик до 3
printf("Значение счетчика: %d. \n", ctr);
ctr = ctr + 1; //увеличивает счетчик до 4
printf("Значение счетчика: %d. \n", ctr);
ctr = ctr + 1; //увеличивает счетчик до 5
printf("Значение счетчика: %d. \n", ctr);
```

```
ctr = ctr - 1; //уменьшает счетчик до 4
printf("Значение счетчика: %d. \n", ctr);
ctr = ctr - 1; //уменьшает счетчик до 3
printf("Значение счетчика: %d. \n", ctr);
ctr = ctr - 1; //уменьшает счетчик до 2
printf("Значение счетчика: %d. \n", ctr);
ctr = ctr - 1; //уменьшает счетчик до 1
return 0;
}
```

Ниже приведено то, как выглядит вывод программы на экране. Обратите внимание, что значение переменной `ctr` продолжает увеличиваться (в компьютерной терминологии такое явление называется *инкрементом* значения) на 1 с каждым выражением присваивания до тех пор, пока оно не станет равным 5 и после чего начинает уменьшаться (*декремент* значения) на 1 с каждым выражением присваивания до тех пор, пока значение счетчика не станет равным 1. (Вычитание значений из счетчика может оказаться полезным, например, если бы вам потребовалось уменьшать количество складских остатков продукции при каждой продаже.)

```
Значение счетчика 1
Значение счетчика 2
Значение счетчика 3
Значение счетчика 4
Значение счетчика 5
Значение счетчика 4
Значение счетчика 3
Значение счетчика 2
Значение счетчика 1
```

В остальных случаях вам потребуется обновить значение переменной, прибавив или иным образом изменив значение итогового показателя. Следующее выражение присваивания увеличивает значение переменной `sales` на 25%:

```
sales = sales * 1.25; /* Увеличить sales на 25% */
```

В языке C предусмотрено несколько *составных операторов*, позволяющих вам обновлять значение переменных методом, похожим на только

что описанный (инкремент, декремент и обновление значения более чем на 1). Однако вместо повторного использования переменной по *обеим* сторонам знака равно, вы можете напечатать значение переменной только один раз. Как и в большинстве случаев, несколько примеров позволят прояснить то, как работают и что делают составные операторы.

ПРИМЕЧАНИЕ

В главе 15 показан более простой способ обновления значения переменной с помощью выражения `for`.

Если вы хотите прибавить 1 к значению переменной, то вы можете воспользоваться оператором *составного сложения* +=. Результат приведенных ниже выражений идентичен:

```
lossCount = lossCount + 1;  /* Прибавляет 1 к lossCount */
lossCount += 1;  /* Прибавляет 1 к lossCount */
```

Вместо умножения переменной `sales` на 1,25 с последующим присвоением ее самой себе:

```
sales = sales * 1.25; /* Увеличить sales на 25% */
```

вы можете воспользоваться оператором *составного умножения*, *=, следующим образом:

```
sales *= 1.25; /* Увеличить sales на 25% */
```

СОВЕТ

Так как вам не нужно печатать имя одной и той же переменной по обеим сторонам знака равенства, гораздо быстрее будет воспользоваться составным оператором, чем расписывать полную форму выражения присваивания. Кроме того, составные операторы позволяют уменьшить количество опечаток в коде, так как вам нет необходимости дважды набирать с клавиатуры имя одной и той же переменной в одном выражении.

В табл. 10.1 приведены все составные операторы присваивания, а также примеры к каждому из них. У всех операторов, от сложения до деления по модулю, которые вы уже встречали в этой книге, есть соответствующий составной оператор.

Табл. 10.1. Составные операторы присваивания

Составной оператор	Пример	Выражение-эквивалент
`*=`	`total *= 1.25;`	`total = total * 1.25;`
`/=`	`amt /= factor;`	`amt = amt / factor;`
`%=`	`days %= 3;`	`days = days % 3;`
`+=`	`count += 1;`	`conut = count + 1;`
`-=`	`quantity -= 5;`	`quantity = quantity - 5;`

Вторая программа-пример выводит на экран абсолютно ту же информацию, что и первая программа в этой главе, однако на этот раз для уменьшения и увеличения значения счетчика используются составные операторы. Кроме того, некоторые составные операторы расположены прямо внутри функции `printf()`, это было сделано с тем, чтобы показать вам, каким образом можно свести две строки кода в одну.

```
//  Программа-пример №2 из главы 10
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter10ex2.c
/* Эта программа увеличивает значение счетчика от 1 до
5, при этом, выводя на печать обновленные значения,
после чего выполняет обратный отсчет до 1. При этом
используются составные операторы */
#include <stdio.h>
main()
{
int ctr = 0;

ctr += 1; //увеличивает счетчик до 1
printf("Значение счетчика: %d. \n", ctr);
ctr += 1; //увеличивает счетчик до 2
printf("Значение счетчика: %d. \n", ctr += 1);
ctr = ctr + 1; //увеличивает счетчик до 4
printf("Значение счетчика: %d. \n", ctr);
printf("Значение счетчика: %d. \n", ctr+1);
ctr -= 1; //уменьшает счетчик до 4
printf("Значение счетчика: %d. \n", ctr);
```

```
printf("Значение счетчика: %d. \n", ctr -= 1);
printf("Значение счетчика: %d. \n", ctr -= 1);
printf("Значение счетчика: %d. \n", ctr -= 1);
return 0;
}
```

Следите за порядком!

Взглянув еще раз на таблицу порядка выполнения математических операций, приведенную в предыдущей главе (табл. 9.1), и найдя в ней составные операторы, вы обнаружите, что у операторов этого типа очень низкий приоритет. Например, оператор += находится на несколько уровней ниже, чем оператор +.

Сначала это может не звучать как нечто важное. (На самом деле, возможно, ничего из того, что мы говорили по этой теме, не кажется вам важным и сложным. Если это так, то *замечательно!* Язык С гораздо легче, чем вам про него рассказывали.) Порядок выполнения операций может встревожить неосведомленного программиста. Подумайте, как подсчитать значение второго выражения:

```
total = 5;
total *= 2 + 3; /* Обновляет переменную total */
```

На первый взгляд может показаться, что значение переменной `total` `13`, так как вы знаете, что умножение выполняется перед сложением. Да, вы правы, умножение на самом деле выполняется перед сложением, но в соответствии с порядком выполнения математических операторов *составное умножение* выполняется *после* сложения. Поэтому сначала вычисляется сумма `2 + 3`, после чего результат сложения (`5`) умножается на старое значение переменной `total` (которое также равняется `5`), соответственно окончательный результат этого выражения равен `25`, как показано на рис. 10.1.

total *= 2+3;

аналогично выражению:

total = total * (2 + 3);

так как *= находится в таблице ниже, чем +

Рис. 10.1. Составные операторы находятся на низком уровне

Приведение типов переменных: Голливуд мог бы поучиться у C спецэффектам

Существует два типа приведения: первый тип относится к спецэффектам киноиндустрии (но мы поговорим с вами не о нем) и второй — приведение типов переменных, использующееся в языке C.

Операция *приведения типа* в языке C временно изменяет тип переменной с одного на другой. Ниже приведен формат операции приведения типа:

```
(типДанных)значение
```

Элемент *`типДанных`* может быть любым типом данных языка C, например `int` или `float`. Элемент *`значение`* — это любая переменная, литерал или выражение. Предположим, что целочисленная переменная `age` содержит значение `6`, следующая строка конвертирует это значение в значение с плавающей точкой `6.0`:

```
(float)age;
```

Если бы вы использовали переменную `age` в выражении совместно с другими переменными типа `float`, то для сохранения целостности выражения вам потребовалось бы привести переменную `age` к типу `float`.

СОВЕТ

При смешивании разных типов данных автоматически могут возникнуть проблемы с округлением значений. Чтобы избежать этих проблем, мы рекомендуем вам применять эксплицитное приведение всех переменных и литеральных констант выражения к единому типу данных.

Не используйте приведение типов в строках кода, когда переменная используется сама по себе. Используйте приведение типов только в том случае, где переменную или выражение необходимо преобразовать для правильного вычисления значения. Предыдущий пример приведения типа переменной `age` может быть проиллюстрирован следующим образом:

```
salaryBonus = salary * (float)age / 150.0;
```

Целочисленная переменная age не превращается в переменную типа float, а лишь *временно приводится* к этому типу только для вычисления значения этого выражения. В остальных участках программного кода, где тип переменной age не приводится эксплицитно, эта переменная так и остается целочисленной.

ВНИМАНИЕ

Если при написании программы вам слишком часто приходится приводить тип какой-либо переменной, то, возможно, вы изначально выбрали неверный тип для этой переменной.

Вы можете выполнить приведение типа для целого выражения. Следующее выражение приводит тип значения выражения перед записью его в переменную:

```
value = (float)(number - 10 * yrsService);
```

Заключение выражения в скобки препятствует приведению типа только переменной number, а не всего выражения. Однако в языке С также предусмотрено автоматическое приведение типов: так, если переменная value объявлена с типом float, то значение выражения автоматически приводится к типу float перед записью в переменную value.

Впрочем, если вы хотите сделать выражения более понятными и не зависеть от автоматического приведения типов, вы можете свободно пользоваться операцией явного (эксплицитного) приведения типов.

Абсолютный минимум

Целью этой главы было научить вас пользоваться дополнительными операторами, помогающими в написании программ на языке С. Кроме того, вы также научились пользоваться приведением типов при смешанном использовании переменных и литеральных констант разного типа. Ниже приведены основные моменты текста этой главы:

- При обновлении значений переменных пользуйтесь составными операторами присваивания
- Используйте составные операторы присваивания для уменьшения количества опечаток и сокращения времени написания программ

- Тип, к которому приводится переменная, выражение или константное значение, заключается в скобки и указывается сразу перед приводимыми переменными, выражениями и константами
- Не смешивайте разные типы данных, вместо этого воспользуйтесь операцией приведения типов, чтобы привести все данные к единому типу перед вычислением значения выражения
- Не следует игнорировать порядок выполнения операторов! Составные операторы имеют очень низкий приоритет и выполняются только после практически всех остальных математических операторов

Глава 11

РАЗВИЛКА НА ДОРОГЕ: ТЕСТИРОВАНИЕ ДАННЫХ ДЛЯ ВЫБОРА ПРАВИЛЬНОГО ПУТИ

В этой главе

- Тестирование данных
- Использование оператора ветвления `if`
- Использование `else`

В языке С предусмотрено очень полезное выражение `if`. Оператор `if` позволяет вашим программам принимать решения и выполнять определенные программные инструкции в зависимости от принятого решения. Тестируя значения переменных, ваша программа может генерировать отличающиеся выходные данные, которые будут зависеть от полученных входных данных.

В этой главе мы также описываем *операторы сравнения*. В сочетании с оператором `if` операторы сравнения превращают язык С в мощный инструмент обработки данных. Компьютеры были бы на самом деле очень скучны, не умей они тестировать данные, более того, если бы компьютеры не могли принимать решения на основе полученных данных, они были бы лишь большими калькуляторами.

Тестирование данных

В языке программирования С выражение `if` (*англ.* «если». — *Примеч. пер.*) работает точно также, как и в разговорной речи: *Если что-то верно, сделать одно действие; иначе сделать другое*. Взгляните на следующие выражения:

```
Если (if) я заработаю денег, мы поедем в Италию.
Если (if) туфли не подходят, вернуть их в магазин.
Если (if) на улице жарко, полить лужайку.
```

В табл. 11.1 перечислены операторы сравнения языка С, позволяющие выполнить тестирование данных. Обратите внимание, что некоторые операторы сравнения состоят из двух символов.

Табл. 11.1. Рациональные операторы языка С

Оператор сравнения	**Описание**
==	Равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно
!=	Неравно

ПРИМЕЧАНИЕ

Операторы сравнения сравнивают два значения. По обеим сторонам оператора всегда необходимо указывать переменную, литеральную константу, выражение или их сочетания.

Прежде чем вы погрузитесь в изучение оператора ветвления `if`, давайте рассмотрим несколько операторов сравнения, чтобы понять, что на самом деле они обозначают. Обычный оператор производит некий математический результат. Оператор сравнения же, в свою очередь, производит результат ИСТИНА или ЛОЖЬ. При сравнении двух значений результатом будет либо подтверждение, либо опровержение. Например, предположим, что переменным присвоены следующие значения:

```
int i = 5;
int j = 10;
int k = 15;
int l = 5;
```

тогда следующие выражения *истинны*:

```
i == l;
j < k;
k > i;
j!= l;
```

Следующие выражения не истинны, значит, они *ложны*:

```
i > j;
k < j;
k == 1;
```

СОВЕТ

Чтобы различать операторы = и ==, запомните, что, дабы утверждать, что что-то равно чему-то, в этом необходимо убедиться дважды.

ВНИМАНИЕ

По обеим сторонам операторов сравнения необходимо помещать только сопоставимые значения, иными словами, не нужно сравнивать символ с числом с плавающей точкой. Если необходимо сравнить два значения разных типов данных, воспользуйтесь операцией приведения типов, чтобы при сравнении оба значения были одинакового типа.

Каждый раз, когда С выполняет оператор сравнения, производится значение `1` или `0`. Истина всегда равна `1`, а ложь всегда равна `0`. Следующие выражения присваивают значение `1` переменной `a` и значение `0` переменной `b`:

```
a = (4 < 10); // (4 < 10) истинно, поэтому в a
записывается 1
b = (5 == 9); // (5 == 9) ложно, поэтому в b
записывается 0
```

В своих программах вы также будете часто пользоваться операторами сравнения, поскольку вам часто будет необходимо знать, превысил ли объем продаж (сохраненный в переменную) поставленную цель, сошелся ли платежный баланс или, например, требуется ли заказать какой-то товар или он еще есть на складе. Сейчас вы видели только верхушку айсберга под названием «операторы сравнения», в следующем разделе мы объясним вам, как ими пользоваться.

Использование оператора ветвления `if`

Для тестирования данных оператор ветвления `if` использует операторы сравнения. Ниже приведен формат выражения с оператором `if`:

```
if (условие)
{ блок из одного или нескольких выражений на языке C; }
```

Заключение `условия` в скобки обязательно. `условие` представляет собой реляционный (сравнительный) тест наподобие тех, о которых шла речь в предыдущем разделе. `блок из одного или нескольких выражений на языке C` также принято называть *телом* выражения с оператором `if`. Фигурные скобки вокруг *блока из одного или нескольких выражений на языке C* обязательны, если тело оператора `if` состоит из нескольких выражений.

СОВЕТ

Даже в том случае, когда фигурные скобки необязательны, то есть оператор `if` содержит в теле только одно выражение, мы рекомендуем все равно использовать фигурные скобки. Скобки вам пригодятся, если в последующем вам потребуется добавить выражения в тело оператора `if`. Если в скобки заключены несколько выражений, то такие выражения, заключенные в скобки, также называются *составным выражением*.

В следующей программе иллюстрируется использование оператора `if`:

```
//  Программа-пример №1 из главы 11
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter11ex1.c
/* Эта программа запрашивает у пользователя год
рождения и вычисляет, сколько лет ему исполняется
в этом году. (Программа также дополнительно проверяет,
не был ли введен еще не наступивший год.) Кроме того,
программа сообщает пользователю, был ли он рожден
в високосный год */
#include <stdio.h>
#define CURRENTYEAR 2013
main()
{
int yearBorn, age;
printf("В каком году Вы родились?\n");
scanf(" %d", &yearBorn);
```

```
// Это выражение if может произвести проверку
// введенных данных, чтобы удостовериться
// в том, что введенный год может
// соответствовать действительности.
// Программа продолжит выполняться только при условии,
// что введенный год наступил раньше текущего года
if (yearBorn > CURRENTYEAR)
{
  printf("Правда? Вы еще не родились?\n");
printf("Хотите попробовать ввести другой год?\n");
printf("В каком году Вы родились?\n");
scanf(" %d", &yearBorn);
}
age = CURRENTYEAR - yearBorn;
printf("\nВ этом году в День рождения Вам исполнится
%d лет!\n", age);
// Второе выражение if использует оператор деления по
// модулю для проверки, не был ли год рождения
// високосным. Этот код будет выполнен только,
// если год окажется високосным
if ((yearBorn % 4) == 0)
{
printf("\nВы родились в високосный год! Круто!\n");
}
return 0;
}
```

Рассмотрим несколько примечаний к программе. Если в своей программе вы используете текущий год, то это хороший пример переменной, которую можно установить с помощью директивы препроцессора `#define`. Таким образом, если вы будете запускать эту программу в будущем, все, что вам понадобиться сделать — изменить одну строку кода.

Первый оператор `if` — это пример того, как выражения с `if` потенциально можно использовать для подтверждения данных. Выражение проверяет, не ввел ли пользователь еще не наступивший год, и, если пользователь ввел неправильный год, программа выполняет участок

кода, помещенный в фигурные скобки. Если же пользователь ввел допустимое значение года, то программа пропускает тело оператора `if` до строки, вычисляющей возраст пользователя. Участок кода в фигурных скобках обращает внимание пользователя на то, что он или она еще не родился (-ась), и дает пользователю возможность ввести другую цифру года. После этого программа продолжает нормальное выполнение.

Здесь вы, возможно, уже заметили серьезный недостаток нашего плана. Если пользователь вводит недопустимое значение года во второй раз, то программа все равно продолжает выполнение и даже сообщает пользователю возраст в виде отрицательного числа! Другой стиль условных выражений, цикл `do ... loop`, продолжает настаивать на введении корректных данных до тех пор, пока пользователь наконец их не введет. Более подробно эта методика описана в главе 14.

СОВЕТ

Вы можете изменить оператор сравнения, позволяющий не принимать введенные данные, если пользователь ввел цифру года больше или равную цифре текущего года, но, возможно, пользователь лишь помогает новорожденному!

После подсчета того, сколько лет исполнится пользователю в этом году в день рождения, выполняется второе выражение `if`, проверяющее с помощью операции деления по модулю год рождения пользователя, и выясняющее, родился ли он (она) в високосный год. Только високосные годы можно нацело разделить на 4, таким образом, только люди, родившиеся в один из таких годов, смогут увидеть сообщение, что они родились в високосный год. Для всех остальных этот участок кода пропускается — и программа достигает точки завершения.

ПРИМЕЧАНИЕ

Покерная программа в приложении Б спрашивает пользователя, какие карты нужно оставить, а какие заменить. Оператор `if` используется для точного определения того, что хочет сделать пользователь.

Иначе...: Использование *else*

В предыдущем разделе вы узнали, как написать алгоритм выполнения программы в случае, если реляционный тест дает результат ИСТИНА.

Если результат теста ЛОЖЬ, ничего не происходит. В этом разделе объясняется использование оператора `else` (*англ.* «иначе». — *Примеч. пер.*), который можно добавить в структуру оператора `if`. С помощью оператора `else` вы можете указать конкретные действия, применяемые программой в случае, если результат реляционного теста ложный. Далее приведен формат комбинированного оператора `if … else`:

```
if (условие)
{ блок из одного или нескольких выражений на языке C; }
else
{ блок из одного или нескольких выражений на языке C; }
```

Таким образом, в случае использования комбинированного оператора `if … else` программа выполнит один из двух участков кода, в зависимости от того, истинно (в этом случае будет выполнен код `if`) или ложно (в этом случае выполняется блок `else`) проверенное условие. Этот оператор подходит идеально, если вы предполагаете два возможных результата и вам нужно выполнить различные программные инструкции для каждого из результатов.

Следующий пример — это обновленная версия предыдущей программы, в которой используется конструкция `if … else`. В этой версии пользователю не предоставляется возможность повторного ввода года рождения, но зато программа поздравляет пользователя с возвращением из будущего.

```
//  Программа-пример №2 из главы 11
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter11ex2.c
/* Эта программа запрашивает у пользователя год
рождения и вычисляет, сколько лет ему исполняется
в этом году. (Программа также дополнительно проверяет,
не был ли введен еще не наступивший год.) Кроме того,
программа сообщает пользователю, был ли он рожден
в високосный год */
#include <stdio.h>
#define CURRENTYEAR 2013
main()
{
int yearBorn, age;
```

```
printf("В каком году Вы родились?\n");
scanf(" %d", &yearBorn);
// Это выражение if может произвести проверку
// введенных данных,
// чтобы удостовериться в том, что введенный год
// может соответствовать действительности.
// Программа продолжит выполняться только при условии, что
// введенный год наступил раньше текущего года
if (yearBorn > CURRENTYEAR)
{
  printf("Правда? Вы еще не родились?\n");
printf("Поздравляем с путешествием во времени!\n");
}
else
{
age = CURRENTYEAR - yearBorn;
printf("\nВ этом году в День рождения Вам исполнится
%d лет!\n", age);
// Второе выражение if использует оператор деления по
// модулю для проверки, не был ли год рождения
// високосным. Этот код будет выполнен только
// если год окажется високосным
if ((yearBorn % 4) == 0)
{
printf("\nВы родились в високосный год! Круто!\n");
}
}
return 0;
}
```

По большому счету, эта программа аналогична предыдущей (за исключением того, что пользователю не дается вторая возможность ввести год рождения, если при первой попытке пользователь ввел недопустимое значение), однако кое-что в этой программе все же достойно вашего внимания. Второе выражение `if` встроено внутрь блока кода `else` пер-

вого оператора `if`. В программировании это принято называть *вложенным выражением*, и вы будете их применять по мере усложнения ваших программ. Вы также можете проверять истинность нескольких условий, в чем вам поможет выражение switch, речь о котором пойдет в главе 17.

СОВЕТ

Употребляйте точку с запятой только в конце исполняемых выражений, находящихся в теле операторов `if` или `else`. Никогда не употребляйте точку с запятой сразу после `if` или `else`, т.к. точка с запятой используется только на конце завершенных выражений.

ПРИМЕЧАНИЕ

Так же как и в теле оператора `if`, использование фигурных скобок в теле оператора `else` не является обязательным в том случае, если тело состоит из одного выражения — одной строки программного кода, — однако использование фигурных скобок является желательным.

В последней программе демонстрируется еще один способ использование операторов `if` и `else`, но на этот раз вы можете проверить истинность четырех различных условий.

```
//  Программа-пример №3 из главы 11
//  Руководства по С для новичков, 3-е издание
//  Файл Chapter11ex3.c
/* Эта программа просит у пользователя оценить свой
уровень счастья по шкале от 1 до 10, а затем выводит
персональное двухстрочное сообщение в зависимости
от уровня счастья 1-2, 3-4, 5-7 или 8-10 баллов*/
#include <stdio.h>
#define CURRENTYEAR 2013
main()
{
int prefer;
printf("Насколько вы счастливы (по шкале от 1 до 10)?\n");
scanf(" %d", &prefer);
// После того, как пользователь ввел свой уровень
// счастья, набор выражений if сопоставляет эти данные
```

```
// с уменьшающимися числами. Только один из четырех
// операторов будет выполнен
if (prefer > = 8)
{
printf("Мы Вас поздравляем!\n");
printf("У Вас все хорошо!\n");
}
else if (prefer > = 5)
{
printf("Чуть выше среднего, не так ли?\n");
printf("Может быть, вскоре все станет еще лучше!\n");
}
else if (prefer > = 3)
{
printf("Сожалеем, что Вам не очень хорошо.\n");
printf("Надеемся, вскоре всё исправится…\n");
}
else
{
printf("Держитесь, ведь все исправится, не так ли?\n");
printf("Перед зарей всегда темнеет.\n");
}
return 0;
}
```

Далее приведен результат двух запусков программы:

```
Насколько вы счастливы (по шкале от 1 до 10)?
5
Чуть выше среднего, не так ли?
Может быть, вскоре все станет еще лучше!
Насколько вы счастливы (по шкале от 1 до 10)?
9
Мы Вас поздравляем!
У Вас все хорошо!
```

Цель этой программы — продемонстрировать, что выражения `if ... else` необязательно должны быть ограничены выбором двух альтернатив. Честно говоря, вы можете создать столько условий `if ... else`, сколько сочтете нужным. Например, вы могли бы установить отдельное сообщение для каждого уровня счастья от 1 до 10. Каждый тест уменьшает количество возможных сценариев выполнения программы, именно поэтому второй тест работает только для чисел от 5 до 7, даже несмотря на то, что программа проверяет введенное число на предмет того, равно ли оно или больше 5. Числа 8 и больше были уже исключены при проверке первым оператором `if`.

Абсолютный минимум

Целью этой главы была демонстрация способов тестирования данных и последующего выполнения того или иного участка кода в зависимости от результатов теста. Не всегда нужно, чтобы при запуске вашей программы разными пользователями выполнялся один и тот же программный код, так как вводимые пользователями данные могут отличаться. Ключевые моменты этой главы перечислены ниже.

- Используйте реляционные операторы для сравнения данных
- Запомните, что результат сравнения ИСТИНА соответствует 1, а ЛОЖЬ — 0.
- Для сравнения данных воспользуйтесь оператором `if`, а также оператором `else` для указания конкретных действий, на случай, если результат сравнения окажется ложным
- Заключайте код в теле выражения `if` и `else` в фигурные скобки. В зависимости от результатов реляционного сравнения, код в фигурных скобках либо выполняется полностью, либо не выполняется вообще.
- После `if` и `else` знак препинания «точка с запятой» не употребляется. Точка с запятой используются только на конце каждого из выражений, включенных в тело операторов `if` и `else`.

Глава 12

ЖОНГЛИРОВАНИЕ ОПЦИЯМИ С ПОМОЩЬЮ ЛОГИЧЕСКИХ ОПЕРАТОРОВ

В этой главе

- Становимся логичными
- Избегаем негатива
- Порядок логических операторов

Иногда реляционные операторы (операторы сравнения), описываемые в главе 11, оказываются просто не в состоянии выразить все условия проверки. Например, если бы вам понадобилось проверить, находится ли числовая или символьная переменная в заданном диапазоне, вам пришлось бы использовать два оператора `if`, например, следующим образом:

```
if (age >= 21) /* Проверка, что 21 <= age <= 65 */
{ if (age <= 65)
{
printf("Возраст попадает в диапазон от 21 до 65.\n");
}
}
```

Хотя в использовании вложенных операторов `if` нет ничего плохого, это не самый прямолинейный метод, кроме того, логика такого кода немного сложнее, чем на самом деле нужно. С помощью *логических операторов*, о которых вы узнаете в этой главе, вы сможете проводить сразу несколько реляционных тестов в одном операторе `if`, что позволит повысить ясность программного кода.

ПРИМЕЧАНИЕ

Не позволяйте терминам «логический» и «реляционный» заставлять вас думать, что операторы, относящиеся к этим группам, сложны. Как только вы поймете, как работает каждый отдельный оператор,

вам не потребуется помнить и постоянно отслеживать, к какой группе относится тот или иной оператор

ПРИМЕЧАНИЕ

Реляционный (сравнительный) оператор просто проверяет, каким образом соотносятся два значения (то есть сравнивает эти значения друг с другом). Логические операторы позволяют сочетать реляционные операторы.

Становимся логичными

Существует три логических оператора (см. табл. 12.1). Иногда логические операторы называют *составными операторами сравнения* (составными реляционными операторами), так как они позволяют сочетать несколько операторов сравнения (см. предыдущее примечание).

Табл. 12.1. Логические операторы

Логический оператор	Значение
`&&`	И
`||`	ИЛИ
`!`	НЕ

Логические операторы употребляются между двумя или более реляционными тестами. Например, ниже приведены первые части трех выражений `if` с использованием логических операторов:

```
if ((age >= 21) && (age <= 65)) {
```

и

```
if ((hrsWorked > 40) || (sales > 25000.00)) {
```

и

```
if (!(isCharterNumber)) {
```

При объединении двух операторов сравнения с помощью логического оператора или использовании оператора `!` (НЕ) для отрицания соответствия, *полностью все* выражение, следующее за оператором `if`, должно быть заключено в скобки. Следующая запись недопустима:

```
if !isCharterNumber { /* Недопустимо */
```

Конечно, предыдущие выражения с `if` содержат больше кода, чем показано выше, но для упрощения примеров мы не приводим код тела оператора.

В программировании логические операторы работают точно так же, как и в разговорной речи. Например, рассмотрим разговорные выражения, соответствующие только что продемонстрированным строкам программного кода:

```
if ((age >= 21) && (age <= 65)) {
```

Это выражение может быть озвучено, например, следующим образом:

«Если возраст как минимум 21 и не больше 65, то…»

Код

```
if ((hrsWorked > 40) || (sales > 25000.00)) {
```

можно превратить в фразу разговорной речи:

«Если количество отработанных часов больше 40 или объем продаж больше 25000р., то…»

Аналогично, выражение

```
if (!(isCharterNumber)) {
```

расшифровывается как

«Если у вас нет номера участника чартерной программы, то…»

Как вы, без сомнения, уже догадались, три приведенные выше выражения устной речи очень точно описывают тесты, производимые операторами `if`. Зачастую в повседневной жизни вы вставляете союз И между двумя условиями, например, «Если вынесешь мусор И приберешься в комнате, то можешь пойти поиграть».

ПРИМЕЧАНИЕ

Перечитайте еще раз строгую фразу, которую вы могли бы сказать ребенку. Условие с союзом И выдвигает жесткое требование, чтобы *оба* задания были сделаны прежде, чем будет получен результат. В языке С тоже самое выполняет оператор `&&`. Тело оператора `if` будет выполнено только в случае, если оба выражения по обеим сторонам оператора `&&` истинны.

Давайте продолжим рассуждать над тем же выражением, только применительно к оператору || (ИЛИ). Вы можете быть более дружелюбным по отношению к ребенку, если скажете следующую фразу: «Если вынесешь мусор ИЛИ приберешься в комнате, то можешь пойти поиграть». Союз «или» менее строгий. Одна из частей (левая или правая) выражения с оператором || должна быть истинна (кроме того, обе части также могут быть истинны). Если одна из частей выражения верна, то будет получен результат. Тоже самое верно и для оператора ||. Одна из частей выражения с оператором || должна быть истинна (истинны могут быть обе части), только в этом случае будет выполнен код, находящийся в теле оператора if.

Оператор ! (НЕ) обращает на противоположное как истинное, так и ложное условие. Истина становится ложью, а ложь — истиной. Звучит запутанно, но так на самом деле и есть! Ограничивайте по возможности использование операторов !. Как правило, можно всегда изменить выражение таким образом, чтобы избежать использования оператора !, просто обратив логику выполнения программного кода. Например, следующее выражение с оператором if:

```
if ( !(sales < 3000)) {
```

абсолютно аналогично вот этому:

```
if (sales >= 3000) {
```

Как вы можете видеть, вы можете превратить отрицательный тест в положительный, убрав оператор ! и изменив оператор сравнения на противоположный.

В следующей программе для проверки данных используются все три логических оператора. Два из трех условий будут выполнены, поэтому на печать будет выведен код из соответствующих блоков оператора if, третье условие не истинно, поэтому на экран будет выведен результат выполнения кода из блока else.

```
//  Программа-пример №1 из главы 12
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter12ex1.c
/* В этой программе определяется набор переменных
и выражений, после чего эти переменные и выражения
сопоставляются между собой с помощью как логических
операторов, так и операторов сравнения. */
```

```
#include <stdio.h>
main()
{
// объявление нескольких общих переменных
int planets = 8;
int friends = 6;
int potterBooks = 7;
int starWars = 6;
int months = 12;
int beatles = 4;
int avengers = 6;
int baseball = 9;
int basketball = 5;
int football = 11;
// В первом логическом выражении используется оператор И
// для проверки, сможет ли актерский состав американского
// комедийного сериала «Друзья» (Friends) и
// участники музыкальной группы Beatles
// составить полноценную бейсбольную команду
// И будет ли основного состава актеров сериала «Друзья»
// и персонажей кинофильма «Мстители» (Avengers)
// достаточно для игры в футбол.
// Если да, то на экран будут выведены соответствующие
// сообщения
if ((friends + beatles >= baseball) &&
(friends + avengers >= football))
{
printf("Актеры \"Друзей\" и участники Beatles ");
printf("могли бы быть бейсбольной командой.\n");
printf("И актеры \"Друзей\" и \"Мстителей\" ");
printf("могли бы стать футбольной командой.\n");
}
else
{
```

```
printf("Актеры \"Друзья\" не смогут стать ");
printf("бейсбольной командой с Фантастической
Четверкой, \n");
printf("ИЛИ они не смогут быть футбольной командой с");
printf("Мстителями (или ни то, ни другое)\n");
}
// Во втором логическом выражении используется
// оператор ИЛИ для проверки, меньше ли кол-во фильмов
// о Звездных войнах числа месяцев в году ИЛИ меньше
// ли количество книг о Гарри Поттере числа месяцев
// в году. Если любое из утверждений верно, то будут
// напечатаны сообщения
if ((starWars <= months) || (potterBooks <= months))
{
printf("\Вы можете читать по книге о Гарри Поттере
в месяц,\n");
printf("и прочитать всю серию менее, чем за год \n");
printf("ИЛИ вы можете смотреть по одному фильму
о Звездных войнах в месяц \n");
printf("и посмотреть всю сагу менее, чем за год.\n");
}
else
{
printf("Ничего из этого не может произойти: слишком
много книг и фильмов,\n");
printf("и слишком мало времени!\n\n");
}
// В последнем логическом утверждении используется
// оператор НЕ для проверки того, НЕ превысит ли сумма
// игроков бейсбольной или баскетбольной команд
// количество игроков футбольной команды
// Если это так, будет напечатано сообщение
if (!(baseball + basketball > football))
{
```

```
printf("\nБейсбольных и баскетбольных игроков в сумме
меньше, чем \n");
printf("футболистов в команде.");
}
else
{
printf("\nБейсбольных и баскетбольных игроков в сумме
больше, чем \n");
printf("футболистов в команде.");
}
return 0;
}
```

Поэкспериментируйте с программой: поизменяйте условия, переменные и операторы, чтобы получить различные комбинации сообщений. Как уже было упомянуто ранее, самый сложный оператор в программе — последний: оператор `!` (НЕ). В большинстве случаев можно написать такое логическое выражение, которое позволяло бы избежать использование этого оператора.

Избегаем негатива

Предположим, что вам нужно написать программу для склада, которая проверяла бы, не подошли ли к нулю запасы определенных товарных позиций. Первая часть оператора `if` может выглядеть следующим образом:

```
if (count == 0) {
```

Так как оператор `if` возвращает значение ИСТИНА только в том случае, если значение переменной count равно `0`, вы можете переписать это выражение следующим образом:

```
if (!count) { /* Тело выполняется только, если count = 0 */
```

Повторимся, оператор `!` несколько запутывает программный код. Даже если вы потратили определенное количество усилий, вводя код со столь привлекательным оператором !, понятный код всегда лучше сложного и запутанного, поэтому лучше использовать вариант `if (count == 0) {`, несмотря на ту микросекунду процессорного времени, которая

экономится благодаря оператору `!`. С помощью оператора `&&` следующая программа выводит на экран сообщение, если фамилия пользователя начинается с букв латинского алфавита в диапазоне от P до S, если же фамилия начинается с других букв, то программа выводит альтернативное сообщение:

```
//  Программа-пример №2 из главы 12
//  Руководства по C для новичков,
// 3-е издание
//  Файл Chapter12ex2.c
/* Эта программа просит пользователя ввести фамилию
латинскими буквами, и если фамилия пользователя
начинается с букв, находящихся в диапазоне от P до S,
его попросят пройти в отдельный кабинет для получения
билетов */
#include <stdio.h>
main()
{
// объявить массив для хранения имени пользователя
// и запросить ввод имени пользователя
char name[25];
printf("Как Ваша фамилия? ");
printf("(Введите с заглавной буквы!)\n");
scanf(" %s", name);
// для символьного массива & не нужен
if ((name[0] >= 'P') && (name[0] <= 'S'))
{
printf("Для получения билетов пройдите ");
printf("в кабинет 2432.\n");
}
else
{
printf("Можете получить билеты здесь.\n");
}
return 0;
}
```

Стоит сделать одно замечание по этой программе. В главе 8 мы предложили вам использовать функцию `printf()` для пояснения того, какие данные и в каком формате должен ввести в программу пользователь. Напоминание пользователям, что они должны ввести свою фамилию с заглавной буквы, позволяет избежать возможных проблем. Если фамилия пользователя `Petrov`, но пользователь ввел ее как `petrov`, с маленькой буквы `p`, то программа не отправит этого пользователя в кабинет 2432, так как логический оператор проверяет только на соответствие диапазону заглавных букв. Теперь, если вы хотите проверить первую букву фамилии на попадание в диапазон как заглавных, так и строчных букв, вы можете воспользоваться более сложным выражением, приведенным ниже:

```
if (((name[0] >= 'P') && (name[0] <= 'S')) || (name[0]
>= 'p') &&(name[0] >= 's')))
```

Это выражение менее удобочитаемо и более сложно для понимания, но такова цена уверенности в правильной обработки введенных данных!

ПРИМЕЧАНИЕ

Отличалось ли бы поведение программы, если бы операторы `&&` были заменены на `||`? Появилось ли бы на экране первое или второе сообщение? Ответ — появилось бы первое сообщение. Программа отправляла бы в кабинет 2432 всех пассажиров. Буквы алфавита от A до Z либо больше P, либо меньше S. Условием теста из предыдущей программы может быть только `&&`, так как доступ в кабинет 2432 есть только у тех людей, имена фамилий которых начинаются с букв, лежащих в диапазоне от P до S.

Как уже упоминалось ранее в этой главе, операторы `if` могут оказаться полезными при проверке правильности ввода информации пользователем. Следующий отрывок программного кода запрашивает у пользователя ответ «да» или «нет» (`Y/N`). В код включен оператор `||`, позволяющий гарантировать, что программе будет передан только допустимый ответ пользователя.

```
printf("Включен ли Ваш принтер (Y/N)?\n");
scanf(" %c", &ans); // перед именем символьной
переменной нужен &
if ((ans == 'Y') || (ans == 'N'))
{
```

```
// выполняется только если пользователь ввел
// корректные данные
if (ans == 'N')
{
printf("*** Включите принтер! ***\n");
}
}
else
{
printf("Вы не ввели Y или N.\n");
}
```

> **СОВЕТ**
>
> С помощью логических операторов вы можете создавать сочетания более чем из двух операторов сравнения, но чересчур большие сочетания могут быть сложными для понимания. Так, например, это выражение избыточно сложное:

```
if ((a < 6) || (c >= 3) && (r!=9) || (p<=1)) {
```

Старайтесь делать так, чтобы создаваемые вами составные операторы сравнения оставались простыми, это повышает удобочитаемость вашего программного кода и упрощает его обслуживание.

Порядок логических операторов

Так как логические операторы также включены в таблицу порядка выполнения операторов, то, как и прочие операторы, они обладают собственными уровнями приоритета. Изучение порядка выполнения логических операторов показывает, что оператор `&&` предшествует оператору `||`. Поэтому компилятор C интерпретирует логическое выражение

```
if (age < 20 || sales < 1200 && hrsWorked > 15) {
```

следующим образом:

```
if ((age < 20) || ((sales < 1200) && (hrsWorked > 15))) {
```

Всегда используйте разделительные скобки, так как они помогают понять порядок выполнения операторов. Компилятор C не запу-

тается, даже если вы не употребите разделительные скобки, так как он отлично знает порядок выполнения операторов. Однако человек, читающий исходный код вашей программы, должен знать, какой оператор выполняется первым, а скобки помогают сгруппировать операторы вместе.

Предположим, что учительница хочет поощрить своих учеников, у которых хорошая успеваемость и очень мало пропусков. Кроме того, для получения поощрения ученик должен либо состоять в трех школьных кружках, либо принимать участие в двух спортивных состязаниях. Уффф! Вы должны признать, что это поощрение будет не только заслуженным, но и сортировка студентов будет очень сложна.

В следующей строке кода на языке C оператор `if` вернет результат ИСТИНА, если ученик соответствует предопределенным учительницей критериям:

```
if (grade > 93 && classMissed <= 3 && numActs >= 3 ||
sports >= 2) {
```

Что ж, это выражение непросто расшифровать. Проблема не только в том, что выражение сложно прочесть, но и в том, что в него вкралась незаметная ошибка. Условие `||` проверяется в последнюю очередь (так как оператор `||` обладает более низким приоритетом по сравнению с оператором `&&`), но это условие `||` должно проверяться перед вторым `&&`. (Если вы уже запутались, это не ваша вина: сложные комбинированные реляционные тесты всегда сложны для восприятия). Далее приведено озвучивание человеческим языком того, как выполняется оператор `if` без разделения частей условия с помощью скобок:

Оценка ученика 93 и более, и ученик пропустил 3 или менее уроков, и принял участие в 3 или более кружках ИЛИ ученик принял участие в 2 или более спортивных состязаниях…

Проблема заключается в том, что для получения поощрения ученику достаточно лишь принять участие в спортивных соревнованиях. Последние две операции сравнения (разделенные оператором `||`) должны сопоставляться перед вторым оператором `&&`. Словесное описание выражения должно выглядеть следующим образом:

Оценка ученика 93 и более, и ученик пропустил 3 или менее уроков и принял участие в 3 или более кружках

и ИЛИ ученик принял участие в 2 ИЛИ более спортивных состязаниях…

Следующее выражение с правильно расставленными скобками делает не только условие оператора `if` более точным, но также более удобочитаемым:

```
if ((grade > 93) && (classMissed <= 3) && ((numActs >=
3) || (sports >= 2)) {
```

Если хотите, то можете разбить такие длинные и сложные выражения с оператором `if` на несколько строк:

```
if ((grade > 93) && (classMissed <= 3) &&
((numActs >= 3) || (sports >= 2)) {
```

Некоторые программисты, работающие с языком С, считают, что два оператора if гораздо проще, чем четыре реляционных теста, результатом такой точки зрения являются, например, следующие выражения:

```
if ((grade > 93) && (classMissed <= 3)
{ if ((numActs >= 3) || (sports >= 2))
{ /* Reward the student */ }
```

Стиль, который вы выберете для себя, зависит от ваших вкусовых предпочтений, от того, какой из вариантов вам кажется наиболее удобным и наиболее простым в обслуживании.

Абсолютный минимум

Целью этой главы было обучить вас использованию логических операторов. Операторы сравнения тестируют данные, логические же операторы `&&` и `||` позволяют вам комбинировать в одном выражении сразу несколько реляционных тестов и выполнять программный код в зависимости от результатов этих тестов. Ниже приведены основные моменты из этой главы.

- Используйте логические операторы для соединения операторов сравнения.
- Используйте оператор `&&`, когда обе части логического оператора должны быть истинны для того, чтобы все выражение считалось истинным.

- Используйте оператор `||` когда одна из двух частей (или обе части) логического оператора должна быть истинна для того, чтобы все выражение считалось истинным.
- Не следует чрезмерно часто использовать оператор `!`. Практически все отрицательные логические выражения могут быть перефразированы, обращены в положительные (таким образом < становится >=, а > превращается в <=). Это позволяет воздержаться от использования оператора НЕ.
- Не следует сочетать слишком большое количество операторов сравнения в одном логическом выражении.

Глава 13

ЕЩЕ МЕШОЧЕК ТРЮКОВ: НОВЫЕ ОПЕРАТОРЫ ДЛЯ ВАШИХ ПРОГРАММ

В этой главе

- До свидания, `if … else`, здравствуй, условный оператор
- Использование операторов небольших изменений: ++ и --
- Примеряем ситуацию

Запаситесь терпением! К настоящему моменту вы уже изучили практически все операторы языка C. Если не брать в расчет несколько более сложных операторов, о которых вы прочитаете в главе 24, эта глава подводит итог изучения таблицы порядка выполнения операторов. В ней пойдет речь об *условных операторах*, *операторах инкремента* и *декремента*.

Операторы языка C зачастую заменяют словесные команды других языков программирования. Большой набор операторов позволяет не только сократить время написания программы, но также повысить эффективность компиляции и выполнения программы по сравнению с командами. Операторы являются очень важным аспектом признанной высокой эффективности языка программирования C.

До свидания, *if … else*, здравствуй, условный оператор

Условный оператор — единственный оператор языка C, которому требуются *три* аргумента. В то время как операторам деления, умножения и большинству других операторов для работы необходимо предоставить два значения, условному оператору необходимо передать три значения. Хотя формат условного оператора на первый взгляд может показаться сложным, вскоре вы убедитесь, что этот оператор упрощает логику и его можно использовать напрямую.

Оператор условия выглядит следующим образом: `?:`. Ниже приведен его формат:

```
отношение ? истинноеВыражение : ложноеВыражение;
```

`Отношение` — это любой реляционный тест, например `age >= 21` или `sales <= 25000.0`. Вы также можете сочетать операторы сравнения с логическими операторами, о которых вы узнали из главы 12. *`истинноеВыражение`* и *`ложноеВыражение`* — это два выражения, написанные в соответствии с правилами языка С. Ниже приведен пример использования оператора условия:

```
(total <= 3850.0) ? (total *= 1.10) : (total *= 1.05);
```

СОВЕТ

Использовать скобки необязательно, но они позволяют сгруппировать три части условного оператора и упростить удобочитаемость такого выражения.

Если тест в первых скобках возвращает значение ИСТИНА, то выполняется *`истинноеВыражение`*, в противном случае, если тест в первых скобках возвращает значение ЛОЖЬ, то будет выполнено *`ложноеВыражение`*. Результат выполнения только что продемонстрированного оператора *абсолютно* аналогичен результату выражения с операторами `if ... else`:

```
if (total <= 3850.0)
{ total *= 1.10; }
else
{ total *= 1.05; }
```

Это выражение сообщает компьютеру, что переменную `total` необходимо умножить на 1,10 или 1,05 в зависимости от результата реляционного теста.

Практически любое выражение с операторами `if ... else` может быть представлено в виде выражения с условным оператором. В этом есть несколько плюсов: условный оператор быстрее вводить с клавиатуры, вы не сможете случайно забыть фигурную скобку, кроме того, условный оператор выполняется более эффективно, чем оператор `if ... else`, так как условный оператор компилируется в более компактный машинный код.

СОВЕТ

Формат условного оператора становится более очевидным, если думать о нем следующим образом: вопросительный знак задает вопрос. Держа этот факт в уме, вы могли бы озвучить предыдущий пример следующим образом: *Переменная total <= 3850.0? Если да, то сделать первое; в противном случае — второе.*

Программисты, работающие на языке C, не любят избыточность условного оператора, которую вы могли заметить в предыдущих примерах. Как можете видеть, переменная `total` появляется в выражении дважды. В обоих случаях этой переменной присваивается определенное значение. Когда вы сталкиваетесь с подобными ситуациями, мы рекомендуем вам избавиться от операции присваивания в выражениях с условным оператором:

```
total *= (total <= 3850.0) ? (1.10) : (1.05);
```

Впрочем, не всегда следует заменять выражение с операторами `if` ... `else` на условный оператор. Зачастую выражения с `if` ... `else` гораздо более удобочитаемы, кроме того, многие условные выражения слишком сложны и их невозможно сжать в условный оператор. Однако, когда необходимо лишь простое выражение ... `else`, условный оператор послужит вам хорошей альтернативой.

У условного оператора по сравнению с оператором `if` есть еще одно дополнительное преимущество: его можно разместить в тех участках кода, куда оператор `if` не может быть помещен. Следующая функция `printf()` выводит на экран окончание –s в случае, если количество съеденных пользователем груш (англ. *pear*) больше одного*:

```
printf("I ate % pear%s\n", numPear, (numPear>1) ?
("s.") : ("."));
```

Если значение переменной `numPear` больше 1, то на экран будет выведено следующее сообщение:

```
I ate 4 pears.
```

* Данный пример не очень хорошо адаптируется на русский язык, так как, в отличие от английского, в русском языке образование множественного числа существительного происходит по более сложной схеме: «я съел 1 грушу, 2 груши, 3 груши, 4 груши, 5 груш, 6 груш» и т. д. Для подбора правильного окончания требуется более сложная структура ветвления с несколькими операторами `if...else` или оператором `switch`. — *Примеч. пер.*

Но, если пользователь съел только одну грушу, то сообщение будет выглядеть следующим образом:

```
I ate 1 pear.
```

ПРИМЕЧАНИЕ

Возможно, вас удивляет, почему условный оператор выглядит как `?:`, но при этом вопросительный знак и двоеточие *никогда* не употребляются вместе. Ну, это так просто потому, что это так. Согласитесь, пояснение наподобие: «условный оператор выглядит как вопросительный знак и двоеточие с каким-то программным кодом между ними».

Далее приведена простая программа, в которой используется условный оператор. (На самом деле этот оператор используется в программе аж восемь раз!) Программа просит пользователя ввести целое число, а затем проверяет, может ли это число быть нацело поделено на все односоставные числа в диапазоне от 2 до 9:

```
//  Программа-пример №1 из главы 13
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter13ex1.c
/* Эта программа просит пользователя ввести число от 1
до 100, а затем сообщает, можно ли это число разделить
нацело на число от 2 до 9. */
#include <stdio.h>
main()
{
// Определить переменную для хранения
// пользовательского числа
// и получить число от пользователя
int numPick;
printf("Выберите число от 0 до 100 ");
printf("(Чем больше, тем лучше!)\n");
scanf(" %d", &numPick); //для переменной типа int,
нужен &
printf("%d %s делится на 2.", numPick, (numPick % 2 ==
0) ? (" ")
```

```
: ("не "));
printf("\n%d %s делится на 3.", numPick, (numPick % 3
== 0) ?
(" ") : ("не "));
printf("\n%d %s делится на 4.", numPick, (numPick % 4
== 0) ?
(" ") : ("не "));
printf("\n%d %s делится на 5.", numPick, (numPick % 5
== 0) ?
(" ") : ("не "));
printf("\n%d %s делится на 6.", numPick, (numPick % 6
== 0) ?
(" ") : ("не "));
printf("\n%d %s делится на 7.", numPick, (numPick % 7
== 0) ?
(" ") : ("не "));
printf("\n%d %s делится на 8.", numPick, (numPick % 8
== 0) ?
(" ") : ("не "));
printf("\n%d %s делится на 9.", numPick, (numPick % 9
== 0) ?
(" ") : ("не "));
return 0;
}
```

ПРИМЕЧАНИЕ

Несмотря на то что выражение с функцией `printf()` запрашивает ввод числа в диапазоне от `1` до `100`, на самом деле вы можете ввести любое целое число. Так, введя число 362880, вы обнаружите, что его можно разделить на все восемь односоставных чисел.

++ и *--*: операторы небольших изменений

Только что изученный условный оператор работает с тремя аргументами, в отличие от него операторы *инкремента* и *декремента* работают только с одним. Оператор инкремента прибавляет 1 к значению пере-

менной, а оператор декремента вычитает 1 из значения переменной. Вот, пожалуй, и все. Почти...

Инкремент и декремент значения переменных — это операции, которые чаще всего выполняются при подсчете чего-либо (например, количество покупателей, совершивших покупки в вашем магазине вчера) или при обратном отсчете (например, подсчет складских остатков при приобретении отдельных позиций покупателями). В главе 10 вы узнали, как увеличивать или уменьшать значения переменных с помощью составных операторов. В этой главе вы изучите операторы, которые позволяют гораздо проще выполнять те же операции. `++` — это оператор инкремента, `--` — оператор декремента. Если вы хотите прибавить 1 к значению переменной `count`, вы можете сделать это следующим образом:

```
count++;
```

Кроме того, вы можете поступить следующим образом:

```
++count++;
```

Оператор декремента выполняет практически то же самое, за исключением того, что 1 вычитается из значения переменной `count`. Этот оператор можно использовать следующим образом:

```
count--;
```

или вот так:

```
--count;
```

Как вы можете видеть, оба оператора могут находиться как слева, так и справа от переменной. Если оператор находится слева, то его называют оператором *преинкремента* или *предекремента*. Если же оператор находится справа, то его обозначают оператором *постинкремента* или *постдекремента*.

ПРИМЕЧАНИЕ

Никогда не применяйте операторы инкремента или декремента по отношению к литеральным константам или выражениям. Операторы инкремента и декремента используются только с переменными. Например, такой записи вы никогда не встретите:

```
--14; /* не делайте так! */
```

Операторы пост/пре- инкремента и декремента дают идентичные результаты, но только когда используются сами по себе. Изменения можно проследить только когда эти операторы используются в составных выражениях. Рассмотрим следующий код:

```
int i= 2, j = 5, n;
n = ++i * j;
```

Вопрос: каково будет значение переменной `n`, когда компьютер завершит выполнение выражений? Несложно угадать, каково будет значение переменной `j`, так как оно остается неизменным, т.е. по-прежнему `j==5`. Оператор ++ позволяет нам быть уверенными в том, что значение переменной `i` будет увеличено на единицу, таким образом, вы знаете, что `i` будет равняться `3`. Если значение переменной `i` увеличивается на единицу перед умножением, то значение переменной n становится равным `15`, в противном случае — (если переменной `i` увеличивается на единицу после умножения) `n==10`.

Ответ кроется в том, является ли оператор ++ оператором преинкремента или постинкремента. Если операторы ++ или `--` находятся *перед* переменной (в префиксе), то компьютер выполнит эти операторы перед выполнением остальных операторов в строке. Если операторы ++ или `--` находятся *после* переменной (в постфиксе), то компьютер выполнит эти операторы после завершения выполнения всех остальных операторов в строке. Так как в предыдущем примере кода оператор ++ находится в положении перед переменной (в префиксе), значение переменной `i` увеличивается на единицу и становится равным `3` прежде, чем выполняется умножение этой переменной на переменную `j`. Следующее выражение увеличивает значение переменной i, делая его равным 3 *после* умножения переменной i на переменную j и записывает ответ в переменную n:

```
n = i++ * j; /*Записывает 10 в n и 3 в i*/
```

Возможность увеличить на единицу значение переменной в том же выражении, в котором вы используете эту же переменную для других вычислений, означает сокращение объемов работы программиста. Предыдущий пример позволяет заменить следующие два выражения, которые вам пришлось бы написать, если бы вы работали с другими языками программирования:

```
n = i * j;
i = i + 1;
```

ПРИМЕЧАНИЕ

Операторы ++ и -- считаются крайне высокоэффективными. Если вам важно (а большинству из нас нет), то операторы ++ и -- компилируются в одно выражение на машинном языке, тогда как прибавление или вычитание единицы с помощью выражений +1 и -1 соответственно не всегда может быть скомпилировано так эффективно.

Давайте еще раз обратимся к программе из главы 10, увеличивающей и уменьшающей значения переменной. На этот раз мы воспользуемся операторами преинкремента и предекремента. Такой подход позволяет значительно сократить количество строк программного кода, если честно, мы сможем сократить объем кода еще сильнее, когда вы изучите циклы в следующей главе.

```
//  Программа-пример №2 из главы 13
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter13ex2.c
/* Эта программа увеличивает значение счетчика от
1 до 5, при этом, выводя на печать обновленные
значения, после чего выполняет обратный отсчет до 1.
Однако в этой версии программы используются операторы
инкремента и декремента */
#include <stdio.h>
main()
{
int ctr = 0;
printf("Значение счетчика: %d. \n", ++ctr);
printf("Значение счетчика: %d. \n", ++ctr);
printf("Значение счетчика: %d. \n", ++ctr);
printf("Значение счетчика: %d. \n", ++ctr);
printf("Значение счетчика: %d. \n", ++ctr);
printf("Значение счетчика: %d. \n", --ctr);
printf("Значение счетчика: %d. \n", --ctr);
printf("Значение счетчика: %d. \n", --ctr);
printf("Значение счетчика: %d. \n", --ctr);
return 0;
}
```

ПРИМЕЧАНИЕ

Чтобы разобраться в отличиях пре- и пост- инкремента и декремента, в предыдущем примере перенесите все операторы инкремента и декремента в положение после переменной `ctr` (`ctr++` и `ctr--`). Угадайте, что произойдет? Скомпилируйте измененную программу — и проверьте, правильно ли вы угадали!

Примеряем ситуацию

Для измерения количество участков памяти, которое потребуется для хранения данных всех типов, используется функция `sizeof()`. Несмотря на то, что сейчас большинство компиляторов языка C используют 4 байта для хранения целочисленных значений, большинство — это не все. Для выяснения того, сколько точно занимают памяти целые числа и числа с плавающей точкой, вы можете воспользоваться функцией `sizeof()`, как показано на примере следующих выражений:

```
i = sizeof(int); // Записывает в i размер целых
значений
f = sizeof(float); // Записывает в f размер дробных
значений
```

Функция `sizeof()` работает как с типами значений, так и с переменными. Если вы хотите выяснить, какой объем памяти занимает конкретная переменная или массив, передайте их в функцию `sizeof()`. В следующем отрывке кода показано, как это сделать:

```
char name[] = "Ruth Claire";
int i = 7;
printf("Размер i %d.\n", sizeof(i));
printf("Размер name %d.\n", sizeof(name));
```

Ниже приведен один из возможных вариантов сообщений, выводимых программой на экран:

```
Размер i 4
Размер name 12
```

В зависимости от конфигурации вашего компьютера и используемого компилятора языка C выводимые сообщения могут отличаться из-за

отличий в размерах целочисленных переменных. Обратите внимание, что размер символьного массива 12, т.к. это значение включает также завершающий строку нуль-символ.

СОВЕТ

Длина строки и размер строки — это два разных понятия. Длина строки — это количество байтов памяти, используемых для ее хранения (не включая нуль-символ), его можно вычислить с помощью функции `strlen()`. Размер строки — это количество символов (знакомест), необходимых для ее хранения, включая нуль-символ.

ПРИМЕЧАНИЕ

Несмотря на то, что сейчас функция `sizeof()` может показаться бесполезной, она вам пригодится позже, по мере изучения языка С.

Абсолютный минимум

Целью этой главы было обобщить ваши знания об операторах языка С. Понимание принципов работы этих операторов не отнимает много времени и не требует многих усилий, но при этом некоторые операторы настолько мощны, что могут заменить целые выражения, используемые в других языках программирования. Далее приведены ключевые концепции этой главы.

- Для повышения эффективности вместо простых выражений if ... else воспользуйтесь условным оператором.
- Условный оператор требует три аргумента. Использование скобок позволяет визуализировать и отделить эти аргументы друг от друга.
- Используйте операторы ++ и `--` для увеличения или уменьшения значения переменной на 1 вместо прибавления или вычитания 1 с помощью присваивания или операторов += и -=.
- Не думайте, что операторы декремента и инкремента, находящиеся в префиксальном или постфиксальном положении, всегда производят одинаковый результат. Эти операторы дают одинаковый результат только если используются с отдельной переменной. Если вы сочетаете операторы ++ и `--` с другими переменными или выражениями, положение этих операторов существенно влияет на получаемый результат.

Глава 14

ПОВТОРЕНИЕ КОДА: ИСПОЛЬЗОВАНИЕ ЦИКЛОВ ДЛЯ ЭКОНОМИИ ВРЕМЕНИ И СИЛ

В этой главе

- Экономия времени благодаря цикличному проходу по программному коду
- Использование конструкции `while`
- Использование конструкции `do...while`

Изучив операторы, вы теперь готовы научиться делать «мертвую петлю» с помощью циклов, включаемых в создаваемые вами программы. *Цикл* — это просто участок кода, выполнение которого повторяется определенное количество раз. Ситуации, когда выполнение цикла не прекращается никогда (так называемый *бесконечный цикл*) следует избегать. Создаваемые вами циклы, в случае если вы указали все параметры верно, по завершение выполнения возложенных на них обязанностей, должны приходить к какому-то выводу.

Зачем вам может понадобиться циклическое выполнение программного кода? Ответ на этот вопрос становится более очевидным, если вспомнить о преимуществе использования компьютеров для выполнения заданий, которые люди вряд ли захотят выполнять. Компьютерам никогда не становится скучно, поэтому мы рекомендуем поручать им рутинные однообразные задания, оставляя задачи, требующие размышления, людям. Навряд ли вы захотите платить работнику, чья единственная обязанность будет вычисление суммы сотен платежных ведомостей, кроме того, очень немногие согласятся выполнять подобную работу. Именно компьютерным программам подходит такая однообразная работа. Когда компьютер завершит цикл программы по подсчету сумм, люди могут приступить к анализу полученных результатов.

Циклы вам могут понадобиться также при необходимости сложения списка чисел, распечатывания финансового итога продаж фирмы за последние 12 месяцев либо подсчета количества студентов, посещающих

дополнительные занятия по информатике. В этой главе пойдет речь о двух широко употребляемых видах циклов языка С, в которых используется команда `while`.

ПОКА мы повторяем

Выражение `while` может появляться как в начале, так и в конце цикла. ПОКА — самый простой вид цикла, в котором используется выражение `while`. (Второй цикл называется «циклом с условием продолжения» `do…while`. Мы дойдем до него чуть позже.) Ниже приведен формат цикла `while`:

```
while (условие)
{блок из одного или нескольких выражений;}
```

Условие — это некий реляционный тест, аналогичный реляционному тесту *`условие`*, с которым вы уже встречались при изучении оператора ветвления `if`. *`Блок из одного или нескольких выражений`* также принято называть *`телом`* цикла `while`.

Выполнение тела цикла `while` продолжается до тех пор, пока *`условие`* верно. В этом заключается основное отличие конструкций `while` и `if`. Тело оператора ветвления `if` выполняется только в том случае, если *`условие`* верно. Тело оператора `if` выполняется только однажды, тогда как тело цикла `while` может выполняться очень много раз.

Рисунок 14.1 помогает объяснить сходства и различия конструкций `while` и `if`. Форматы обеих команд похожи даже в том, что необходимо использовать фигурные скобки в случае, если тело цикла `while` состоит из нескольких выражений. Однако тело цикла `while` следует заключать в фигурные скобки, даже если оно состоит из единственного выражения: при добавлении дополнительных выражений в тело цикла пропадает необходимость отвлекаться на расстановку соответствующих фигурных скобок. Никогда не употребляйте точку с запятой (`;`) после круглых скобок, в которые заключается условие цикла `while`. Точкой с запятой следует завершать только выражения внутри тела цикла `while`.

ВНИМАНИЕ

Несмотря на то что два выражения на рис. 14.1 похожи, результат их выполнения не будет одинаков. Операторы `while` и `if` — две самостоятельные конструкции, предназначенные для выполнения разных задач.

Значение переменной, помещенной в `условие` цикла `while`, *должно* измениться, в противном случае цикл `while` будет выполняться бесконечно, так как компьютер будет проводить проверку истинности одного и того же `условия` при каждом новом прохождении цикла.

Избежать бесконечных циклов можно, удостоверившись, что при выполнении цикла `while` что-то меняется в его `условии` таким образом, что со временем `условие` становится ложным — и выполнение программы продолжается исполнением программных инструкций, следующих после цикла `while`.

```
Использование if
if (amount < 25)
{
  printf("Значение слишком мало.\n"); ← Выполняется
  wrongVal++;                           только однажды,
}           ПРОДОЛЖАТЬ ВЫПОЛНЕНИЕ       но при условии,
                                        что значение
                                        amount меньше 25.

Использование while
while (amount < 25)
{
  printf("Значение слишком мало.\n "); ← Выполнение
  wrongVal++;                             повторяется
  printf("Попробуй снова…Каково новое значение?\n);   до тех пор,
  scanf("%d", &amount);                   пока значение
}                       ВЕРНУТЬСЯ         amount
                                          меньше 25
```

Рис. 14.1. Тело оператора `if` выполняется однажды, тело цикла `while` может выполняться более одного раза

ПРИМЕЧАНИЕ

Как и в случае с оператором `if`, тело цикла `while` может быть не выполнено ни разу. Если уже при первом вхождении в цикл `условие` ложно, тело цикла `while` выполнено не будет.

Использование цикла *while*

Если вам необходимо, чтобы выполнение определенного участка программного кода повторялось несколько раз, вы можете добиться этого с помощью цикла `while`. Давайте еще раз вернемся к программе, перечисляющей числа, но только в этот раз воспользуемся циклами `while`:

```
//  Программа-пример №1 из Главы 14
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter14ex1.c
/* Эта программа увеличивает значение счетчика
от 1 до 5, при этом выводя каждое новое значение
на экран, после чего программа отсчитывает значение
обратно к 1. Однако в этой версии используются циклы
while, а также операторы инкремента и декремента */
#include <stdio.h>
main()
{
int ctr = 0;
while (ctr < 5)
{
printf("Значение счетчика %d.\n", ++ctr);
}
while (ctr > 1)
{
printf("Значение счетчика %d.\n", ++ctr);
}
return 0;
}
```

Возможно, вам уже очень надоел наш пример с изменением «значения счетчика», но использование разных выражений, форматов и функций для решения одной и той же задачи — отличный метод демонстрации того, как новые навыки помогут вам по-иному, возможно, эффективнее решить поставленную задачу.

При сравнении листинга из этого примера с листингами программных кодов предшествующих примеров решения этой задачи вы можете видеть, как сильно сократился объем программного кода благодаря использованию циклов. Раньше вам приходилось набирать пять выражений `printf()` для увеличения значения счетчика, а потом еще пять — для его уменьшения. Однако использование циклов `while` требует написания только одного выражения `printf()` для увеличения значения счетчика и еще одного — для его уменьшения. Такой подход позволяет повысить эффективность программирования.

Изначально значение переменной `ctr` равно `0`. При первичном выполнении цикла `while` значение переменной `i` меньше `5`, поэтому пока *условие* истинно программа продолжает выполнять тело цикла `while`. В теле цикла производится вывод на экран строки текста и увеличение на единицу значения переменной `ctr`. При втором входе в цикл тестируется *условие*, значение переменной `ctr` равно `1`, 1 все еще меньше 5, поэтому тело цикла выполняется вновь. Выполнение тела цикла повторяется до тех пор, пока значение переменной `ctr` не будет увеличено до `5`. Так как `5` не меньше 5 (числа равны), *условие* становится ложным и повторение цикла прекращается. Таким образом расчищается дорога для выполнения остального кода программы, что приводит ко второму циклу `while`, который отвечает за обратный отсчет от 5 до 1. Когда с течением времени *условие* второго цикла становится ложным, выполнение этого цикла прекращается.

СОВЕТ

Если бы в теле цикла `while` не происходило постепенное увеличение значения переменной `ctr`, функция `printf()` выполнялась бы бесконечное количество раз, до тех пор, пока вы не нажали бы сочетание клавиш **Ctrl+Break** для экстренной остановки программы.

Использование цикла *do...while*

Цикл `while` также можно использовать в сочетании с выражением `do`. При таком совместном использовании обе конструкции обычно называют выражением `do...while` или циклом `do...while`. Ниже приведен формат цикла `do...while`:

```
do
{блок из одного или нескольких выражений;}
while (условие)
```

ПРИМЕЧАНИЕ

Хотя выражения `do` и `while` играют роль своего рода рамок для тела цикла, использование фигурных скобок обязательно, если тело цикла состоит более чем из одной строки кода.

Используйте цикл `do...while` вместо обычного цикла `while` только в том случае, если тело цикла *должно быть выполнено хотя бы один*

раз. Так как *условие* расположено только в конце цикла do...while, компьютер не имеет возможности проверить истинность этого условия, пока тело цикла не будет выполнено в первый раз.

Ниже приведена небольшая программа, в которой применяется цикл do...while. Эта программа просит пользователя ввести два числа, а затем выводит результат их умножения. После чего программа спрашивает, не желает ли пользователь произвести умножение следующей пары чисел, и до тех пор, пока пользователь вводит ответ *Y*, программа продолжает запрашивать ввод чисел для умножения. Только ответ *N* прерывает повторение цикла.

```
//  Программа-пример №2 из главы 14
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter14ex2.c
/* Программа выполняет умножение двух чисел и выводит
результат умножения на экран, пока этого хочет
пользователь. Ввод ответа N прервет выполнение цикла
*/
#include <stdio.h>
main()
{
float num1, num2, result;
char choice;
do {
printf("Введите первый множитель: ");
scanf(" %f", &num1);
printf("Введите второй множитель: ");
scanf(" %f", &num2);
result = num1 * num2;
printf("%.2f умножить на %.2f равняется %.2f\n\n",
num1, num2, result);
printf("Выполнить умножение ");
printf("следующей пары чисел? (Y/N): ");
scanf(" %c", &choice);
//Если пользователь в качестве ответа ввел символ n
//нижнего регистра, то этот оператор условия if
```

```
//преобразует его в символ N верхнего регистра
if(choice == 'n')
{
choice = 'N';
}
} while (choice != 'N');
return 0;
}
```

Несмотря на то, что программа очень проста и прямолинейна, это пример эффективного использования цикла `do...while`. Повторимся, что конструкцию `do...while` нужно использовать вместо цикла `while` в том случае, если вы хотите быть уверенным, что тело цикла точно выполнится хотя бы один раз. Поэтому после получения от пользователя двух чисел с плавающей точкой программа спрашивает пользователя, не желает ли он повторить операцию умножения для двух других чисел. Если пользователь вводит ответ *Y* (или любой другой символ, кроме *N*), выполнение цикла начинается сначала.

Если бы в цикл не было включено выражение с оператором ветвления `if`, то введение пользователем буквы `n` нижнего регистра также привело бы к выходу из цикла, впрочем, кажется вполне очевидно, что пользователь, вводящий ответ — символ `n` нижнего регистра — также желал бы завершить выполнение цикла, но просто забыл нажать клавишу **Shift**. Как мы уже писали ранее в этой книге, при создании программ не всегда следует полагаться на то, что пользователь будет всегда вводить именно те данные, которые вы от него запрашиваете.

Таким образом, по возможности рассматривайте несколько общих случаев ошибочного ввода и предоставляйте механизмы их обработки. Преобразование символа `n` нижнего регистра в символ `N` верхнего — не единственный способ решения проблемы. Например, вы также могли бы воспользоваться логическим оператором И в участке кода, задающем условие повторения цикла `while`:

```
} while (choice != 'N' && choice != 'n');
```

Говоря простым языком, эта программная инструкция сообщает компьютеру продолжать выполнение программы до тех пор, пока значение переменной `choice` не равно символу `n` нижнего или символу `N` верхнего регистра.

СОВЕТ

В главе 19 объясняется более простой метод проверки на соответствии введенных символов символам `Y` и `N` верхнего регистра или `y` и `n` нижнего с помощью встроенной функции `toupper()`.

Абсолютный минимум

Целью этой главы было продемонстрировать способы повторного выполнения заданных участков программного кода. Циклы `while` и `do...while` предназначены для многократного выполнения программного кода, помещенного в тело этих циклов. Разница между этими циклами заключается в местоположении реляционного теста, контролирующего выполнение цикла. Цикл `while` проверяет истинность отношения вверху цикла, а цикл `do...while` — внизу, благодаря чему выражения, находящиеся в теле этого цикла обязательно выполняются хотя бы один раз. Ниже перечислены основные моменты повествования этой главы:

- Если необходимо несколько раз выполнить отдельный участок кода, воспользуйтесь циклом `while` или `do...while`
- Убедитесь, что при выполнении выражений из тела циклов `while` и `do...while`, в *условии* цикла происходят определенные изменения: в противном случае, созданный вами цикл будет повторяться бесконечное количество раз
- Запомните, что от оператора ветвления `if` циклы отличаются тем, что в случае, если *условие* истинно, тело оператора `if` выполняется только один раз, тогда как тело цикла — несколько раз.
- Не используйте знак «точка с запятой» после закрывающей скобки условия выражения `while`, иначе вы ненароком создадите бесконечный цикл.

Глава 15
ЕЩЕ ОДИН СПОСОБ СОЗДАНИЯ ЦИКЛОВ

В этой главе

- Поиск новых способов повторного выполнения кода
- Работа с циклом for

Еще один тип цикла, предусмотренного в языке С, называется циклом ДЛЯ (`for`). Цикл `for` предоставляет больше контроля по сравнению с циклами `while` и `do...while`. Используя цикл `for`, вы можете указать конкретное количество повторений цикла, тогда как в случае с циклами `while` вы должны повторять выполнение программного кода вновь и вновь, пока условие не станет ложным.

В программах, написанных на языке С, есть место для циклов всех трех типов. Иногда один тип цикла больше подходит для использования в одной программе, а другой тип — в другой программе. Например, если бы вы создавали программу для обработки заказов клиентов, приобретающих определенные товары со склада, то вам потребовался бы цикл `while`. Такая программа обрабатывала бы заказы, ПОКА клиенты все еще заходят в двери вашего магазина. Если ваш магазин посетило 100 клиентов, то цикл `while` будет выполнен 100 раз. В конце дня для подсчета дневной выручки вам, возможно, потребуется сложить суммы покупок всех 100 клиентов. Для этой цели вам подойдет цикл `for`, так как вы точно знаете, сколько раз следует повторить выполнение цикла.

ПРИМЕЧАНИЕ

Увеличивая значения переменных-счетчиков, вы можете симулировать цикл `for` с помощью цикла `while`. Кроме того, вы также можете симулировать цикл `while` с помощью цикла `for`! Таким образом, тип выбираемого вами цикла целиком и полностью зависит от того, с каким циклом вам будет удобнее работать.

ДЛЯ и во имя повторения!

Как можете видеть из глупого названия раздела, цикл `for` важен для контроля за повторным выполнением участков кода. Формат цикла `for` немного необычен:

```
for (начальноеВыражение; контрольноеВыражение; счетчик)
{ блок из одного или нескольких выражений; }
```

Возможно, все прояснит пример с работающим программным кодом:

```
for (ctr = 1; ctr <= 5; ctr++)
{
  printf("Значение счетчика %d.\n", ctr);
}
```

Если на первый взгляд код из примера показался вам несколько знаком, то вы абсолютно правы. Этот код будет уже пятой версией программы, постепенно увеличивающей и затем уменьшающей значение счетчика. Отличие в том, что в этой версии мы используем цикл `for`. Выражение `for` работает следующим образом: при входе в цикл `for` выполняется *начальноеВыражение*, то есть, `ctr = 1;`. В любом цикле `for` *начальноеВыражение* выполняется ***только один раз***. После этого производится проверка истинности *контрольногоВыражения*. В данном примере *контрольноеВыражение* — `ctr <= 5;`. Если данное выражение истинно, — а в данном примере в первый раз оно точно будет истинно, — происходит выполнение тела цикла `for`. По завершении выполнения тела цикла выполняется выражение-*счетчик*, то есть производится инкремент значения переменной `ctr`.

> **СОВЕТ**
>
> Как можете видеть, сдвиг тела цикла `for` вправо позволяет его визуально отделить от остального кода программы, что делает код более удобочитаемым (это утверждение верно и для других типов циклов, например, цикла `do…while`).

Пожалуй, информации уже слишком много: ее так просто не переварить, хотя это всего лишь один абзац текста. Давайте упростим эту задачу. Проследите глазами за линией на рис. 15.1, она показывает поря-

док выполнения выражений цикла `for`. Следя за линией, перечитайте предыдущий абзац: это должно упростить вам его понимание.

```
for(ctr=1;ctr <=10;ctr++)

  {printf("Все еще считаю...);

   printf("%d.\n", ctr);


   }
```

Рис. 15.1. Порядок выполнения программных инструкций цикла `for`

ПРИМЕЧАНИЕ

Формат цикла `for` может показаться несколько странным из-за обязательных к употреблению символов «точка с запятой» прямо в строке кода. Действительно, символ «точка с запятой» употребляется только на конце исполняемых выражений, однако выражения внутри цикла `for` *являются* исполняемыми. В частности, как показано на рис. 15.1, выполнение начального выражения `ctr = 1;` завершается еще до начала выполнения цикла.

Ниже приведен тот же самый цикл, только записанный с помощью выражения `while`:

```
ctr = 1;
while (ctr <= 5)
{
  printf("Значение счетчика %d.\n", ctr);
ctr++;
}
```

Далее приведен экранный вывод указанного выше программного кода:

```
Значение счетчика 1
Значение счетчика 2
Значение счетчика 3
Значение счетчика 4
Значение счетчика 5
```

> **СОВЕТ**
>
> Если вы вновь проследите за направляющей линией на рис. 15.1 и прочтете предыдущий пример с циклом `while`, то вы увидите, что оба цикла `while` и `for` выполняют одинаковую работу. Выражение `ctr =1;`, предшествующее циклу `while`, — это первое выражение, выполняемое в цикле `for`.

Цикл for не может быть в полной мере представлен циклом `do...while`, так как реляционный тест производится *перед* выполнением тела цикла for и только после выполнения тела цикла `do...while`. Как вы можете вспомнить из текста главы 14, тест цикла `do...while` всегда находится в конце цикла.

Работа с циклом `for`

Цикл `for` (ДЛЯ) можно прочитать фразами из ежедневного обихода. Рассмотрим следующее выражение:

```
Для каждого из 45 сотрудников рассчитать зарплату
и распечатать чек.
```

В этом выражении совсем нет места для двусмысленного восприятия. Будет 45 сотрудников, 45 операций вычисления заработной платы и печать 45 чеков. Чтобы сделать этот цикл пригодным для большего количества компаний, программа может запрашивать пользователя ввести количество сотрудников, для которого будет производиться расчет зарплаты, после чего введенные пользователем данные могут быть использованы для цикличного повторения операций, например следующим образом:

```
printf("Сколько сотрудников в вашей организации? ");
scanf(" %d", &employees);
//Цикл для подсчета з/п каждого из сотрудников
for (i=1; i<=employees; i++)
{
  //Расчет зарплаты для каждого сотрудника
```

Циклы `for` могут быть использованы не только для увеличения значения счетчика, как было показано в предыдущих двух примерах. Цикл `for`, приведенный далее, *уменьшает* значение счетчика перед печатью сообщения:

```
for (cDown=10; cDown>0; cDown--)
{
  printf("%d.\n", cDown);
}
printf("Взрыв!\n");
```

Далее приведен экранный вывод данного программного кода:

```
10
9
8
7
6
5
4
3
2
1
Взрыв!
```

ВНИМАНИЕ

Если последнее выражение в скобках цикла `for` уменьшает значение счетчика, то для выполнения тела цикла изначальное значение должно быть больше контрольного значения. В предыдущем примере начальное значение `10` больше значения `0`, сравнение с которым производится *контрольнымВыражением* цикла `for`.

Вовсе необязательно увеличивать или уменьшать счетчик цикла на 1. Следующий цикл `for` увеличивает счетчик сразу на 3, начиная с 1:

```
for (i = 1; i < 18; i += 3)
{
  printf("%d ", i); //Печатает 1, 4, 7, 10, 13, 16
}
```

Следующий код производит интересный эффект:

```
for (outer = 1; outer <= 3; outer++)
```

```
{
for (inner = 1; inner <= 5; inner++)
{
printf("%d ", inner);
}
//Печатает новую строку каждый раз, когда
заканчивается //вложенный цикл
printf("\n");
}
```

Далее приведен экранный вывод кода:

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

Помещение цикла в тело другого цикла называется *вложением* цикла. В результате вложенный (внутренний) цикл выполняется столько раз, сколько указывает внешний цикл. Вложенный цикл мог бы вам понадобиться, например, если бы вам было необходимо трижды напечатать список пяти лучших клиентов. Значение счетчика внутреннего цикла переходило бы от `1` до `3`, в то время как внутренний цикл распечатывал бы список пяти лучших клиентов.

Далее приведена полнофункциональная программа, выполняющая циклы `for` столько раз, сколько фильмов просмотрел пользователь в этом году. Программа запрашивает название фильма и его рейтинг по шкале от 1 до 10. По получении всех оценок программа сообщает пользователю, какой из фильмов был оценен как любимый, а какой — как наименее понравившийся.

```
//  Программа-пример №1 из главы 15
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter15ex1.c
/* Эта программа спрашивает пользователей, сколько
фильмов они видели в этом году, а затем входит в цикл,
спрашивая название каждого фильма и его оценку по
шкале от 1 до 10. Программа запомнит самый любимый
и наименее понравившийся фильм*/
#include <stdio.h>
```

```
#include <string.h>
main()
{
int ctr, numMovies, rating, favRating, leastRating;
char movieName[40], favourite[40], least[40];
//инициализировать favRating значением 0,
//т.о. любой фильм
//с рейтингом 1 или выше заменит
//это значение
//и leastRating
//значением 10, т.о. любой фильм
//с рейтингом 9 и ниже
//заменит его
favRating = 0;
leastRating = 10;
//Выяснить, сколько фильмов посмотрел и может оценить
//пользователь
//Цикл будет выполняться, пока пользователь не введет
//значение больше 0
do {
printf("Сколько фильмов вы посмотрели за этот год? ");
scanf(" %d", &numMovies);
//Если пользователь вводит 0 или отрицательное число,
//программа напомнит ввести положительное число
//и вновь выведет на экран запрос
if (numMovies < 1)
{
printf("Ни одного фильма! Как вы можете их оценивать?
\nПопробуйте снова!\n\n);
}
} while (numMovies <1);
for (ctr = 1; ctr <= numMovies; ctr++)
{
//Получить название фильма и оценку пользователя
```

```
printf("\nВведите название фильма ");
printf("(название только в 1 слово!) ");
scanf(" %s", movieName);
printf("Как бы вы его оценили по шкале");
printf("от 1 до 10? ");
scanf(" %d", &rating);
//Проверить, не имеет ли данный фильм наивысший
//рейтинг
if (rating > favRating)
{
strcpy(favorite, movieName);
favRating = rating;
}
//Проверить, не имеет ли данный фильм низший рейтинг
if (rating < leastRating)
{
strcpy(least, movieName);
leastRating = rating;
}
}
printf("\nВаш любимый фильм: %s.\n", favorite);
printf("\nВаш наименее любимый фильм: %s.\n", least);
return 0;
}
```

Далее приведен возможный экранный вывод программы:

```
Введите название фильма (название только в 1 слово!)
Veranda
Как бы вы его оценили по шкале от 1 до 10? 7
Введите название фильма (название только в 1 слово!)
Easiness
Как бы вы его оценили по шкале от 1 до 10? 3
Введите название фильма (название только в 1 слово!)
TheJuggler
```

```
Как бы вы его оценили по шкале от 1 до 10? 5
Введите название фильма (название только в 1 слово!)
Kickpuncher
Как бы вы его оценили по шкале от 1 до 10? 8
Введите название фильма (название только в 1 слово!)
Celery
Как бы вы его оценили по шкале от 1 до 10? 8
Ваш любимый фильм: Kickpuncher
Ваш наименее любимый фильм: Easiness
```

Программа из этого примера несколько длиннее предыдущих, но вы должны уже смочь проследить ход ее выполнения строчка за строчкой, в этом вам также помогут комментарии. В программе объединяется использование циклов `for` и `do...while`, а также реализовано несколько проверок данных с помощью выражений `if`. Первое выражение `if` выполняет роль тестера данных. Вы спрашиваете пользователей, сколько фильмов они посмотрели, после чего программа циклически выполняет участок кода, чтобы получить название и рейтинг для каждого просмотренного фильма. При этом количество повторений соответствует введенному пользователем количеству просмотренных фильмов. Если пользователь ввел `0` (или по ошибке было введено отрицательное число), то цикл выполнен не будет, поэтому с помощью цикла `do...while` вы даете пользователю шанс ввести корректные данные.

На первый взгляд операции присваивания значения `0` переменной `favRating` и `10` переменной `leastRating` может показаться непонятным, но при входе в цикл для получения названий фильмов и рейтингов вам потребуется некий ориентир, с которым вы бы сравнивали рейтинг каждого фильма.

Изначально любимому фильму необходимо присвоить самое низкое значение из возможных, таким образом, что любое оцененное кино автоматически становилось бы любимым. Это значит, что первый же фильм (в примере программного кода — фильм «Veranda») станет одновременно и любимым, и наименее понравившимся фильмом одновременно. Однако в этом есть смысл, только если вы видели всего лишь одно кино: оно будет для вас одновременно и лучшим, и худшим до тех пор, пока не появится нечто, с чем это кино можно будет сравнить.

При вводе в систему дополнительных фильмов два выражения `if`, вложенных в цикл, проверят, нравится ли вам это кино больше или меньше текущего любимого и наименее понравившегося фильмов соответственно. По результатам проверки будут внесены соответствующие изменения.

В примере второй фильм — «Easiness» — имеет рейтинг 3. Этот рейтинг не выше рейтинга фильма «Veranda» (7), поэтому фильм «Veranda» остается предпочитаемым, однако наименее понравившемся фильмом теперь становится «Easiness».

По мере изучения языка программирования C вы научитесь справляться с некоторыми проблемами, присутствующими в этой программе. Первая проблема — это ограничения функции `scanf()` при работе со строками. Дело в том, что данная функция может принимать только одно слово без пробелов. Очевидно, что названия большинства фильмов состоят из нескольких слов. Эту проблему вы сможете исправить, изучив дополнительные методы ввода/вывода информации позже в этой книге.

Когда вы изучите другие массивы, в том числе массивы указателей, вы сможете хранить названия всех фильмов в программе наподобие этой. Вы также научитесь сортировать данные, таким образом, вы сможете отредактировать программу и распечатать ранжированный список просмотренных фильмов, отсортировав их от самого понравившегося до наименее понравившегося. Такой список позволит вам решить еще одну проблему, так как данная версия программы может хранить только одно название фильма для каждой оценки, поэтому если пользователь вводит две одинаковые оценки (например, для фильмов Kickpuncher и Celery), только один из этих фильмов может быть обозначен как любимый.

Абсолютный минимум

Целью данной главы было показать вам дополнительный способ создания циклов на языке программирования C. Выражение `for` дает вам несколько больше контроля над выполнением цикла, нежели выражения `while` и `do…while`. Выражение `for` контролирует выполнение цикла с помощью переменной, инициализируемой и изменяемой в соответствии с выражениями внутри объявления цикла `for`. Далее перечислены основные концепции данной главы:

- Для инкремента или декремента значения переменной с помощью цикла воспользуйтесь циклом `for`.
- Запомните, что реляционный тест цикла `for` выполняется в самом начале цикла.
- Воспользуйтесь вложенным циклом, если вы хотите выполнить цикл заданное количество раз.
- Не забывайте символы «точка с запятой» внутри объявления цикла `for`: для цикла `for` их использование обязательно.
- Если необходимо, чтобы цикл `for` совершил обратный отсчет значений счетчика, не задавайте начальное значение меньше контрольного значения.

Глава 16
ВХОД И ВЫХОД ИЗ ЦИКЛИЧНОГО КОДА

В этой главе

- Делаем перерыв на кофе-брейк
- Продолжаем работать

В этой главе вы не узнаете о том, как пользоваться еще одной разновидностью циклов. Вместо этого данная глава призвана расширить информацию, которую вы уже изучили при прочтении предыдущих двух глав. Так, для цикла `while` доступны дополнительные средства контроля за исключением уже известного вам реляционного теста, кроме того, вы можете изменить ход выполнения цикла `for` средствами, отличными от счетчика. Выражения `break` и `continue` позволяют вам контролировать ход выполнения циклов в тех особых случаях, когда вам нужно преждевременно покинуть цикл или повторить выполнение цикла раньше, чем он бы повторился автоматически.

Делаем перерыв на кофе-брейк

Выражение `break` очень редко, даже, скорее всего, никогда не встречается само по себе. Как правило, это выражение можно встретить в теле оператора ветвления `if`. Вскоре мы проясним вам причину этого. Далее приведен формат выражения `break`:

```
break;
```

ПРИМЕЧАНИЕ

Выражение `break` имеет очень простой формат, не правда ли? Действительно, это так. Однако следует помнить, что, как правило, выражение `break` находится внутри оператора `if`. Таким образом, оператор `if` — это своего рода первая часть практически всех встречающихся выражений `break`.

Выражение break всегда появляется внутри какого-либо цикла. Цель выражения break — прекратить выполнение текущего цикла. По завершении цикла управление ходом выполнения программы переходит к коду, следующим за этим циклом. Когда в теле цикла используется выражение break, оно немедленно прекращает выполнение цикла, и управление передается коду, следующему сразу за прерванным циклом.

Далее приведен цикл for, который в нормальных условиях вывел бы на экран 10 чисел, однако вместо вывода 10 чисел на экране будут напечатаны только 5, так как выполнение цикла прерывается выражением break:

```
for (i=0; i < 10; i++)
{
printf("%d ", i);
if (i == 4)
{
break;
}
}
// Продолжение программного кода
```

В качестве следующего примера возьмем задачу из реального мира. Предположим, преподаватель составил программу для подсчета среднего балла 25 студентов. Программа, приводимая далее, получает баллы 25 студентов, однако если один или два студента пропустили тест, то преподавателю уже не нужно подсчитывать среднюю оценку из расчета на 25 студентов.

Если в качестве оценки преподаватель вводит –1.0, то введенное значение работает как триггер, провоцируя срабатывание выражения break — и выполнение цикла прекращается.

```
//  Программа-пример №1 из главы 16
//  Руководства по С для новичков, 3-е издание
//  Файл Chapter16ex1.c
/* Данная программа запрашивает у пользователя ввод
оценок за тест 25 студентов группы и рассчитывает
средний балл. Если тест писали менее 25 студентов,
```

```
то пользователь может ввести -1 вместо оценки, тогда
программа выйдет из цикла и для подсчета среднего
балла будут использованы только введенные оценки */
#include <stdio.h>
main()
{
int numTest;
float stTest, avg, total = 0.0;
//Запрос ввода максимум 25 оценок
for (numTest = 0; numTest < 25; numTest++)
{
//Получение оценки и проверка ввода триггера -1
printf("\nВведите оценку студента: ");
scanf(" %f", &stTest);
if (stTest < 0.0)
{
break;
}
total += stTest;
}
avg = total / numTest;
prtintf("\n Средний балл %.1f%%.\n", avg);
return 0;
}
```

Прежде чем обсуждать программу, давайте взглянем на результат ее тестового запуска:

```
Введите оценку студента: 89.9
Введите оценку студента: 92.5
Введите оценку студента: 51.0
Введите оценку студента: 86.4
Введите оценку студента: 78.6
Введите оценку студента: -1
Средний балл 79.7%
```

Очень уж много студентов этого преподавателя заболели в день тестирования! Если бы на занятия пришли все 25 студентов, то цикл for гарантировал бы запрос всех 25 оценок. Но так как на тест пришли лишь 5 студентов, преподаватель должна была сообщить программе, что ввод оценок закончен, введя в нашем случае отрицательное значение, таким образом программа «узнала», что преподаватель желает уже сейчас получить средний балл за тест.

СОВЕТ

Для печати математического символа «процент» при выводе среднего бала в функции `printf()` мы использовали два символа `%`. Если не продублировать символ `%`, как это было сделано в программе-примере, компьютер воспримет данный символ как управляющую последовательность. Впрочем, компьютер все равно интерпретирует первый символ как символ управления для второго символа. Иными словами, знак `%` — управляющий символ сам для себя.

ПРЕДУПРЕЖДЕНИЕ

Выражение `break` лишь позволяет произвести преждевременный выход из циклов `while`, `do...while` и `for`, однако данное выражение не может выйти из оператора `if`, так как данный оператор не является оператором цикла. Рисунок 16.1 помогает показать работу выражения `break`.

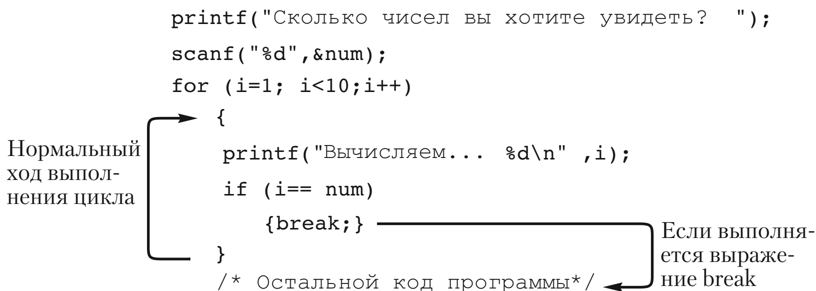


Рис. 16.1. Выражение break прерывает выполнение цикла

Давайте продолжим работать

Выражение `break` приводит к преждевременному *прекращению* цикла, выражение `continue`, в свою очередь, заставляет цикл пре-

ждевременно *продолжиться*. (Что, в принципе, соответствует значению глаголов «break» и «continue» в английском языке.) В зависимости от сложности созданного вами цикла `while`, `do...while` или `for`, возможно, вы не захотите, чтобы *тело цикла выполнялось целиком при каждой итерации*. Выражение `continue` как бы говорит компьютеру: «Пожалуйста, в этот раз не обращай внимание на оставшуюся часть этого цикла, вернись к началу — и начни заново».

СОВЕТ

Итерация — это модное словечко из компьютерного лексикона, обозначающее один проход цикла. Некоторые программисты склонны думать, что если никто из окружающих не будет понимать употребляемых ими слов, то они уж точно не потеряют работу.

Следующий пример очень хорошо иллюстрирует использование выражения `continue` на практике. Программа содержит цикл `for`, считающий от 1 до 10. Если переменная цикла содержит нечетное число, то на экран выводится сообщение «От нечетных чисел одни неудобства...», а выражение `continue` сообщает компьютеру проигнорировать оставшийся код цикла, так как выполнение этого кода выводит на экран сообщение «То ли дело четные числа!» для четных значений переменной.

```
//  Программа-пример №2 из главы 16
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter16ex2.c
/* Данная программа выполняется циклически для
10 чисел и выводит на печать сообщение, зависящее
от того, является ли обрабатываемое число четным.
Программа проверяет, не является ли число
нечетным, и, если число нечетное, выводит на экран
соответствующее сообщение для нечетных чисел, после
чего начинает новую итерацию цикла с помощью выражения
continue. В противном случае, на экран выводится
сообщение для четных чисел. */
#include <stdio.h>
main()
{
```

```
int i;
//Проход по числам от 1 до 10
for (i = 1; i <=10; i++)
{
if ((i%2) == 1) //Остаток от деления нечетных чисел =
1
{
printf("От нечетных чисел одни неудобства…\n");
// Перескочить к началу новой итерации цикла
continue;
}
printf("То ли дело четные числа! \n");
}
return 0;
}
```

Далее приведен результат выполнения данной программы:

```
От нечетных чисел одни неудобства…
То ли дело четные числа!
От нечетных чисел одни неудобства…
То ли дело четные числа!
От нечетных чисел одни неудобства…
То ли дело четные числа!
От нечетных чисел одни неудобства…
То ли дело четные числа!
От нечетных чисел одни неудобства…
То ли дело четные числа!
```

ПРИМЕЧАНИЕ

Как и в случае с выражением `break`, выражение `continue` очень редко используется без какого-либо предшествующего оператора ветвления `if`. Если бы вам было необходимо *постоянно* продолжать (continue), вы бы не вводили последнюю часть тела цикла. Использовать выражение continue вам потребуется только при некоторых проходах цикла.

Абсолютный минимум

Целью данной главы было научить вас более эффективным методикам контроля выполнения циклов: выражениям `break` и `continue`. Все виды циклов (`while`, `do...while`, `for`) могут быть преждевременно прекращены с помощью выражения `break` и продолжены — с помощью выражения `continue`. Основные моменты текста главы перечислены далее.

- Для преждевременного выхода из циклов воспользуйтесь `while`, `do...while` и `for` выражением `break`.
- Чтобы начать новую итерацию цикла, воспользуйтесь выражением `continue`.
- Не пользуйтесь выражениями `break` и `continue` без какого-либо реляционного теста, предшествующего им.

Глава 17
КЕЙС ДЛЯ ПЕРЕКЛЮЧАТЕЛЯ

В этой главе

- Тестирование нескольких `case` с помощью переключателя `switch`
- Сочетание выражений `break` и `switch`

Оператор ветвления `if` хорош для проведения простых проверок данных, особенно в том случае, если проверка данных предусматривает два-три варианта развития событий. Конечно, вы можете использовать оператор `if` для тестирования на соответствие более чем двух значений, однако в этом случае вам придется вложить друг в друга несколько операторов `if`, а это может привести к путанице и сложностям в поддержке программы. Представьте на минуту, как бы вы составили программный код, основанный на реакции пользователя на какое-либо меню — список опций для выбора, например меню может выглядеть следующим образом:

```
Что бы вы хотели сделать?
1. Добавить новый контакт
2. Редактировать существующий контакт
3. Вызвать контакт
4. Написать текстовое сообщение контакту
5. Удалить контакт
6. Покинуть программу
Каков ваш выбор?
```

ПРИМЕЧАНИЕ

Создавая меню, запрашивающие ввода данных от пользователя, вы создаете пользовательский интерфейс.

Для обработки все предложенных опций вам потребовалось бы использовать пять вложенных операторов `if`, как продемонстрировано ниже.

```
if (userAns ==1)
{
//Процедура добавления контакта
}
else if (userAns == 2)
{
  //Процедура редактирования контакта
}
else if (userAns == 3)
{
  //Процедура вызова контакта
}
else if (userAns == 4)
{
  //Процедура отправки сообщения контакту
}
else if (userAns == 5)
{
  //Процедура удаления контакта
}
else
{
  //Процедура выхода из программы
}
```

В общем, во вложенных операторах ветвления `if` нет абсолютно ничего плохого, однако выражение `switch` языка C гораздо более понятное и удобочитаемое, когда речь заходит о выборе из множества вариантов.

Поворачиваем переключатель

Формат выражения `switch`, пожалуй, самый длинный из форматов всех остальных выражений языка C (или любого другого языка программирования). Далее приведен формат выражения `switch`:

```
switch (выражение)
{
case (выражение1): {одно или несколько выражений C; }
case (выражение1): {одно или несколько выражений C; }
case (выражение1): {одно или несколько выражений C; }
//Таких блоков может быть столько, сколько вам нужно
default: {одно или несколько выражений C; }
}
```

СОВЕТ

Как и с большинством выражений, на самом деле пользоваться выражением `switch` гораздо проще, чем может показаться на первый взгляд.

Меню, продемонстрированное ранее, идеально подходит для серии вызовов функций. Проблема лишь в том, что в этой книге мы еще не обсуждали вызовы функций, за исключением горстки встроенных функций, таких как `printf()` и `scanf()`. В следующей простой программе используется выражение `switch` для вывода соответствующего текстового сообщения на экран, в зависимости от выбора, сделанного пользователем.

СОВЕТ

Обычно в каждом из блоков case функция `printf()` будет заменена на вызов соответствующей функции. После прочтения главы 31 вы будете знать, как использовать вызовы функций для обработки блоков `case`.

```
//  Программа-пример №1 из Главы 17
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter17ex1.c
/*Программа предоставляет пользователю меню опций
и принимает выбор пользователя, после чего использует
выражение switch для выполнения одной-двух строк
кода в соответствии с выбором пользователя.
(Реализации выполняемых действий нет, данная
программа представляет собой просто набор заглушек,
демонстрирующих ценность выражения switch)*/
```

```
#include <stdio.h>
#include <stlib.h>
main()
{
int choice;
printf("Что бы вы хотели сделать?\n");
printf("1. Добавить новый контакт\n");
printf("2. Редактировать существующий контакт\n");
printf("3. Вызвать контакт\n");
printf("4. Написать текстовое сообщение контакту\n");
printf("5. Удалить контакт\n");
printf("6. Покинуть программу\n");
do
{
printf("Сделайте выбор:\n");
scanf(" %d", &choice);
switch (choice)
{
case(1): printf("\nЧтобы добавить контакт");
  printf(" вам понадобится ввести\n");
  printf("Фамилию, имя и номер телефона\n");
  break;
case(2): printf("\nПриготовьтесь ввести");
  printf(" имя контакта,\n");
  printf("который нужно изменить\n");
  break;
case(3): printf("\nКакой контакт ");
    printf(" вы хотели бы вызвать?\n");
    break;
case(4): printf("\nКакому контакту ");
    printf(" вы хотели бы написать?\n");
    break;
case(5): exit(1); //Ранний выход из программы
default: printf("\n%d неправильный выбор.\n",
```

```
choice);
printf("Попробуйте снова.\n");
break;
}
} while ((choice < 1) || (choice > 5));
return 0;
}
```

Выражения `case` определяют ход выполнения программы в зависимости от значения переменной `choice`. Например, если значение переменной `choice` равняется 3, то на экран будет выведено сообщение «`Какому контакту вы хотели бы позвонить?`». Если же значение переменной `choice` равняется 5, то встроенная функция `exit()` осуществит выход из программы.

ВНИМАНИЕ

Каждый раз, когда вам необходимо преждевременно завершить выполнение программы, вы можете воспользоваться функцией `exit()`. Значение, помещаемое вами между скобок функции `exit()`, передается операционной системе. Большинство начинающих программистов не обращают внимание на возвращаемое значение и просто помещают в скобки `0` или `1`. Запомните, что для того, чтобы воспользоваться функцией `exit()`, в программу необходимо подключить заголовочный файл `<stdlib.h>` с помощью директивы `#include`.

Цикл `do...while` заставляет пользователя сосредоточиться на программе. Если пользователь вводит данные, отличные от числа в диапазоне между 1 и 5, то благодаря ключевому слову `default` программа выведет на экран сообщение «`...неправильный выбор`». Язык с гарантирует, что если введенные данные не соответствуют ни одному из прописанных в теле оператора `switch` условий, то будет выполнено выражение `default`.

Можно сказать, что выполнено выражение `default` чем-то напоминает оператор `else`: именно оператор `else` принимает управление, если тест оператора `if` оказывается ложным. Аналогично, выражение `default` также принимает управление, если ни одно из прочих условий не соответствует значению переменной, переданной оператору `switch`. Хотя выражение `default` не является строго обязательным (точно так

же, как и оператор `else`), использование выражения `default` является хорошей практикой программирования по обработке неожиданных значений переменных, передаваемых оператору `switch`.

СОВЕТ

Оператору `switch` может быть передана либо целочисленная, либо символьная переменная. Не проверяйте значения переменных типа `float` или `double` с помощью оператора `switch`.

break и switch

В предыдущем примере в коде оператора `switch` вы могли заметить несколько выражений `break`. Это выражение выступает своего рода гарантом того, что программа выполнит только один блок выражений `case`. Без выражения `break`, оператор `switch` «сорвется» и пройдет по всем оставшимся блокам выражений `case`. Далее приведен пример того, что бы произошло, если бы мы убрали все выражения `break` и пользователь выбрал бы пункт меню под номером `2`:

```
Приготовьтесь ввести имя контакта,
который нужно изменить
Какой контакт вы хотели бы вызывать?
Какому контакту вы хотели бы написать?
```

Выражение `break` предотвращает одновременное выполнение сразу нескольких блоков выражений `case`.

ПРИМЕЧАНИЕ

В данном примере не было выполнено выражение `default`, но это произошло только лишь потому, что программа была преждевременно завершена с помощью функции `exit()` в блоке `case(5)`.

Размышления об эффективности

Нет необходимости пытаться организовать блоки `case` в каком бы то ни было порядке. Даже блок `default` необязательно должно быть последним. Более того, если блок `default` все же находится в самом конце оператора `switch`, то употреблять выражение `break` в этом

блоке также необязательно. Однако использование выражения `break` в блоке `default` позволяет вам быть уверенными, что вы перемещаете оба выражения в случае, если вам когда-либо понадобится изменить порядок следования блоков `case`. Если бы вы поместили блок `default` выше одного, нескольких или всех блоков `case`, то для предотвращения выполнения блоков `case` использование в блоке `default` выражения `break` было бы обязательным.

СОВЕТ

Для повышения эффективности вы можете попытаться изменить порядок следования блоков `case`. Помещайте наиболее возможные варианты как можно ближе к началу, чтобы компьютеру не приходилось проверять каждый блок в поисках нужного.

Давайте создадим вторую программу, с помощью которой мы продемонстрируем использование оператора `switch` и создание программы с двумя уровнями меню.

```
//  Программа-пример №1 из главы 17
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter17ex2.c
/* Данная программа предоставляет меню опций (три
десятилетия), принимает выбор пользователя, а затем
выводит на экран меню второго уровня (спорт,
развлечения, политика).
Когда пользователь осуществляет второй выбор,
программа выводит на экран основной информационный
список, соответствующий этому десятилетию */
#include <stdio.h>
#include <stdlib.h> //Запомните, что если планируете
         // воспользоваться функцией exit(),
//вам нужен этот заголовочный файл
main()
{
//Despite being a long program, you only need two
variables:
//one for the first menu and one for the second
int choice1;
```

```
int choice2;
//Варианты десятилетий на выбор
printf("Что бы вы хотели увидеть?\n");
printf("1. 1980-е\n");
printf("2. 1990-е\n");
printf("3. 2000-е\n");
printf("4. Выход\n");
//Выбор пунктов верхнего меню и выражение switch,
//выводящее информацию на экран
//а также цикл do-while, гарантирующий, что
//осуществлен выбор одного из 4 пунктов меню
do
{
printf("Введите свой выбор: ");
scanf(" %d", &choice1);
switch(choice1)
{
//В первом случае пользователь выбрал 1980-е.
//Теперь узнаем, что конкретно он хотел бы //увидеть.
case (1):
{
printf("\n\nКакой раздел бы вы хотели увидеть\n");
printf("1. Бейсбол\n");
printf("2. Кино\n");
printf("3. Президенты США\n");
printf("4. Выход\n");
printf("Введите свой выбор: ");
scanf(" %d", &choice2);
if (choice2 == 1)
{
printf("\n\nЧемпионы мировой серии ");
printf("1980-х годов:\n");
printf("1980: Philadelphia Phillies\n");
printf("1981: Los Angeles Dodgers\n");
```

```
printf("1982: St. Louis Cardinals\n");
printf("1983: Baltimore Orioles\n");
printf("1984: Detroit Tigers\n");
printf("1985: Kansas City Royals\n");
printf("1986: New York Mets\n");
printf("1987: Minnesota Twins\n");
printf("1988: Los Angeles Dodgers\n");
printf("1989: Oakland A’s\n");
printf("\n\n\n");
break;
} else if (choice2 == 2)
{
printf("\n\nОбладатели премии Оскар ");
printf("1980-х годов:\n");
printf("1980: Ordinary People\n");
printf("1981: Chariots of Fire\n");
printf("1982: Gandhi\n");
printf("1983: Terms of Endearment\n");
printf("1984: Amadeus\n");
printf("1985: Out of Africa\n");
printf("1986: Platoon\n");
printf("1987: The Last Emperor\n");
printf("1988: Rain Man\n");
printf("1989: Driving Miss Daisy\n");
printf("\n\n\n");
break;
} else if (choice2 == 3)
{
printf("\n\nПрезиденты США ");
printf("1980-х годов:\n");
printf("1980: Джимми Картер\n");
printf("1981-1988: Рональд Рейган\n");
printf("1989: Джордж Буш\n");
printf("\n\n\n");
```

```
break;
} else if (choice2 == 4)
{
exit(1);
} else
{
printf("Извините, неправильный номер!\n");
break;
}
}
//Этот блок case для 1990-х годов.
//В отличие, от верхнего меню, проверка
// выбранного пункта меню не предусмотрена
case (2):
{
printf("\n\nКакой раздел бы вы хотели увидеть\n");
printf("1. Бейсбол\n");
printf("2. Кино\n");
printf("3. Президенты США\n");
printf("4. Выход\n");
printf("Введите свой выбор: ");
scanf(" %d", &choice2);
if (choice2 == 1)
{
printf("\n\nЧемпионы мировой серии ");
printf("1990-х годов:\n");
printf("1990: Cincinnati Reds\n");
printf("1991: Minnesota Twins\n");
printf("1992: Toronto Blue Jays\n");
printf("1993: Toronto Blue Jays \n");
printf("1994: No World Series\n");
printf("1995: Atlanta Braves\n");
printf("1996: New York Yankees\n");
printf("1997: Florida Marlins\n");
```

```
printf("1998: New York Yankees\n");
printf("1999: New York Yankees\n");
printf("\n\n\n");
break;
} else if (choice2 == 2)
{
printf("\n\nОбладатели премии Оскар ");
printf("1990-х годов:\n");
printf("1990: Dances with Wolves\n");
printf("1991: The Silence of the Lambs\n");
printf("1992: Unforgiven\n");
printf("1993: Schindler's List\n");
printf("1996: The English Patient\n");
printf("1997: Titanic\n");
printf("1998: Shakespeare in Love\n");
printf("1999: American Beauty\n");
printf("\n\n\n");
break;
} else if (choice2 == 3)
{
printf("\n\nПрезиденты США ");
printf("1990-х годов:\n");
printf("1990-1992: Джордж Буш\n");
printf("1993-1999: Билл Клинтон\n");
printf("\n\n\n");
break;
} else if (choice2 == 4)
{
exit(1);
} else
{
printf("Извините, неправильный номер!\n");
break;
}
```

```
}
//Этот участок кода задействуется, когда
//пользователь выбирает 2000-е годы
case (3):
{
printf("\n\nКакой раздел бы вы хотели увидеть\n");
printf("1. Бейсбол\n");
printf("2. Кино\n");
printf("3. Президенты США\n");
printf("4. Выход\n");
printf("Введите свой выбор: ");
scanf(" %d", &choice2);
if (choice2 == 1)
{
printf("\n\nЧемпионы мировой серии ");
printf("2000-х годов:\n");
printf("2000: New York Yankees \n");
printf("2001: Arizona Diamondbacks\n");
printf("2002: Anaheim Angels\n");
printf("2003: Florida Marlins\n");
printf("2004: Boston Red Sox\n");
printf("2005: Chicago White Sox\n");
printf("2006: St. Louis Cardinals\n");
printf("2007: Boston Red Sox\n");
printf("2008: Philadelphia Phillies\n");
printf("2009: New York Yankees\n");
printf("\n\n\n");
break;
} else if (choice2 == 2)
{
printf("\n\nОбладатели премии Оскар ");
printf("2000-х годов:\n");
printf("2000: Gladiator\n");
printf("2001: A Beautiful Mind\n");
```

```
printf("2002: Chicago\n2003: The");
printf(" Lord of the rings: The ");
printf("Return of the King\n");
printf("2004: Million Dollar Baby\n");
printf("2005: Crash\n");
printf("2006: The Departed\n");
printf("2007: No Country for Old Men\n");
printf("2008: Slumdog Millionaire\n");
printf("2009: The Hurt Locker\n");
printf("\n\n\n");
break;
} else if (choice2 == 3)
{
printf("\n\nПрезиденты США ");
printf("2000-х годов:\n");
printf("2000: Bill Clinton\n");
printf("2001-2008: George Bush\n");
printf("2009: Barrack Obama\n");
printf("\n\n\n");
break;
} else if (choice2 == 4)
{
exit(1);
} else
{
printf("Извините, неправильный номер!\n");
break;
}
}
case (4):
exit(1);
default:
printf("Пункта меню \n%d не существует.\n",choice1);
printf("Try again.\n");
```

```
break;
}
} while ((choice1 < 1) || (choice1 > 4));
return 0;

}
```

Несмотря на то, что на первый взгляд данная программа может показаться достаточно сложной и запутанной, но давайте подумаем вот над чем: во-первых, вы прочитали уже большую половину данной книги, поэтому вам пора обрести уверенность в собственных познаниях в языке С. Во-вторых, длинная программа не значит сложная: просто разделите код и проанализируйте код по секциям — и вы поймете, что в этом коде нет ничего сложного.

В программе предусмотрено два уровня меню. В меню верхнего уровня вы просите пользователя выбрать интересующее его десятилетие: 1980-е, 1990-е или 2000-е. После того, как пользователь вводит `1`, `2` или `3`, в соответствии с выбранным десятилетием, (или `4` — для выхода из программы), оператор `switch` переводит программу на следующий уровень меню. Пользователь может получить информацию о спорте (в частности, о бейсболе), кинофильмах или президентах США. Внутри каждого блока case предусмотрен оператор ветвления `if...else`, проверяющий введенные пользователем данные для корректного отображения запрошенной информации.

Возможно, вы задаетесь вопросом: «Хм, ведь использование оператора `switch` на верхнем уровне меню было хорошей идеей, почему бы не использовать его также и для обработки выбираемых пользователем пунктов меню следующего уровня?». Следует начать с того, что хотя вы можете вложить операторы `if` друг в друга, вложение операторов `switch` не очень хорошая идея, потому как в этом случае выражения `default` начинают накладываться друг на друга, это запутывает компьютер — и программа не выполняется.

Следует также заметить, что в первом примере мы не использовали открывающие и закрывающие фигурные скобки в блоках `case`, но использовали их во втором примере. В данном случае фигурные скобки не являются обязательными, но их использование с более сложными блоками кода позволяет прояснить и повысить удобочитаемость программного кода.

СОВЕТ

Вы сможете заменить повторяющиеся участки кода меню второго уровня всего лишь одной строкой, когда по прочтению книги научитесь создавать собственные функции

Абсолютный минимум

Целью данной главы было объяснить использование выражения `switch` языка C. Выражение `switch` анализирует значение целочисленной или символьной переменной и выполняет один из нескольких участков кода, называемых блоками `case`. Вы можете написать эквивалентный по выполняемым задачам код с помощью оператора `if`, но использование выражения `switch` делает код более понятным, особенно когда речь идет о реакции программы (анализ и выполнение соответствующего блока кода) на выбор пользователем определенного пункта меню. Далее перечислены основные концепции данной главы.

- Используйте оператор `switch` для создания кода программного меню или кода других приложений, осуществляющих выбор одного из множества значений.
- С оператором `switch` используйте целочисленные или символьные значения, так как правильно соотнести значения типа `double` и `float` весьма затруднительно.
- Если хотите избежать выполнения последующих блоков `case`, сопровождайте каждый блок `case` выражением `break`.
- Если возможно использовать оператор `switch`, старайтесь избегать использование вложенных операторов `if`: код оператора `switch` гораздо более удобочитаем.

Глава 18
УСОВЕРШЕНСТВОВАНИЕ ВВОДА И ВЫВОДА ВАШИХ ПРОГРАММ

В этой главе

- Использование функций `putchar()` и `getchar()`
- Размышления о новых строках
- Небольшое ускорение с помощью функции `getch()`

Кроме функций `scanf()` и `printf()` существуют еще другие способы для осуществления операций ввода и вывода.

В этой главе мы покажем вам несколько встроенных функций ввода-вывода языка C, которые вы сможете использовать для управления вводом-выводом (I/O). Вы можете использовать эти простые функции для создания собственных мощных процедур ввода данных.

Функции, о которых пойдет речь, предоставляют *примитивные* возможности для ввода или вывода за раз лишь одного символа.

Разумеется, для отдельных символов вы также можете пользоваться спецификатором `%c` при работе с функциями `scanf()` и `printf()`, однако функции ввода-вывода символов, о которых речь пойдет далее, более просты в использовании, а также предоставляют несколько возможностей, которых лишены функции `scanf()` и `printf()`.

Функции *putchar()* и *getchar()*

Функция `getchar()` принимает с клавиатуры один символ, тогда как функция `putchar()` посылает один символ на экран. На рис. 18.1 показано, что происходит при использовании этих функций. В целом они работают так, как вы и предполагаете. Вы можете ими воспользоваться в любое время, если хотите вывести на печать или ввести один символ в переменную.

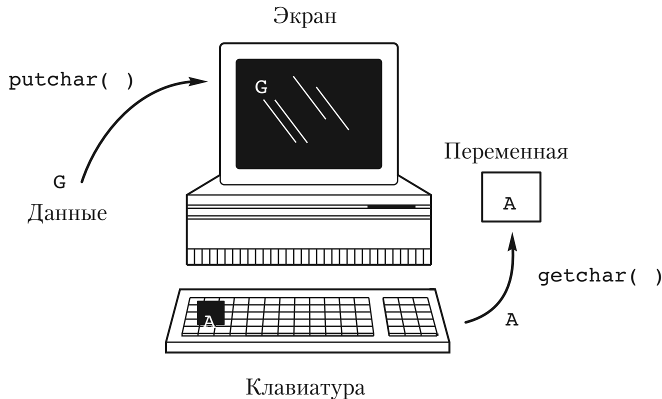


Рис. 18.1. Функции `getchar()` и `putchar()` используются для ввода и вывода одного символа

> **СОВЕТ**
>
> При использовании функций ввода-вывода из этой главы не забывайте подключать к своей программе заголовочный файл `stdio.h`: тот же самый файл, который вы подключаете при работе с функциями `scanf()` и `printf()`.

По-английски название функции `getchar()` созвучно с фразой «Get character», что переводится как «взять символ», а название функции `putchar()` — с фразой «Put character» («поместить символ»). Что ж, похоже, разработчики языка C знали что делали!

Следующая программа выводит на экран сообщение «`C - это весело`», при этом фраза выводится посимвольно с помощью функции `putchar()`, по порядку распечатывающей каждый отдельный элемент символьного массива. Обратите внимание на использование функции `strlen()`, гарантирующей, что цикл `for` не будет пытаться распечатать элементы, следующие после последнего символа в строке.

```
//  Программа-пример №1 из главы 18
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter18ex1.c
/*Эта программа — лишь простая демонстрация функции
putchar() */
//функция putchar() определена в файле stdio.h, однако
// файл string.h нужен для работы функции strlen
#include <stdio.h>
```

```
#include <string.h>
main()
{
int i;
char msg[] = "C - это весело";
for (i = 0; i < strlen(msg); i++)
{
putchar(msg[i]); //Вывод одного символа
}
putchar('\n'); //Перенос строки после завершения цикла
return(0);
}
```

Функция `getchar()` возвращает символ, введенный с клавиатуры. Таким образом вы, как правило, либо присваиваете символ переменной, либо производите с ним какие-либо действия. Вы также можете использовать функцию `getchar()` саму по себе, например:

```
getchar(); /*никаких действий над символом не
производится*/
```

Однако большинство компиляторов языка С предупредят вас о бесполезности данного выражения. Функция `getchar()` получит символ, введенный с клавиатуры, но дальше никаких действий над этим символом произведено не будет. Далее приведена программа, принимающая по одному символу и сохраняющая полученные символы в символьный массив. После этого набор функций `putchar()` выводит массив в обратном порядке.

```
//  Программа-пример №2 из главы 18
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter18ex2.c
/*Эта программа — лишь простая демонстрация функции
getchar() */
//функция getchar() определена в файле stdio.h, однако
// файл string.h нужен для работы функции strlen
#include <stdio.h>
#include <string.h>
main()
{
int i;
```

```
char msg[25];
printf("Введите макс. 25 символов и нажмите Enter…\n");
for (i = 0; i < 25; i++)
{
msg[i] = getchar(); //Вводит один символ
if (msg[i]=='\n')
{
i--;
break;
}
}
putchar('\n'); //Перенос строки после завершения цикла
for (; i >= 0; i--)
{
putchar(msg[i]);
}
putchar('\n');
return(0);
}
```

ПРИМЕЧАНИЕ

Обратите внимание, что во втором цикле `for` у переменной `i` нет начального значения. На самом деле оно у нее есть. Переменная `i` содержит номер последнего индекса массива, введенного с помощью функции `getchar()` в предыдущем цикле `for`. Таким образом, второй цикл начинает работу со значения переменной `i`, оставленного предшествующим циклом `for`.

Обычно символ, вводимый функцией `getchar()` определяется типом `int`, как было сделано в предыдущем примере. Целочисленный и символьный типы — это, пожалуй, единственные типы данных языка C, которые могут заменять друг друга без проблем с приведением типа. В некоторых более сложных приложениях функция `getchar()` может возвращать значения, не работающие в типе `char`, поэтому, используя тип `int`, вы в полной безопасности. Разве вы не счастливы, что уже изучили выражение `break`? Программа продолжает принимать по одному символу до тех пор, пока пользователь не нажмет клавишу **Enter** (нажатие которой производит управляющую последовательность `\n`), после чего выражение `break` останавливает выполнение цикла.

Размышления о новых строках

Не смотря на то, что функция `getchar()` принимает только один символ, управление ходом выполнения программы не возвращается к вам до тех пор, пока пользователь не нажмет клавишу **Enter**. На самом деле функция `getchar()` инструктирует компьютер принимать ввод в *буфер*, или участок памяти, зарезервированный для хранения вводимых данных. Освобождения буфера не происходит до тех пор, пока пользователь не нажмет клавишу **Enter**, после чего содержание буфера освобождается посимвольно. Это имеет два эффекта. Во-первых, до тех пор, пока не была нажата клавиша **Enter**, пользователь может воспользоваться клавишей **Backspace** для корректировки введенных данных. Во-вторых, факт нажатия клавиши **Enter** сохраняется в буфере, если вы не удалите его вручную.

Устранение факта нажатия клавиши **Enter** — это одна из проблем, с которыми, должно быть, сталкиваются все начинающие программисты.

Для этой проблемы существует несколько решений, но ни одно из них нельзя назвать элегантным. Рассмотрим следующий участок программного кода:

```
printf("Введите первые буквы имени и фамилии: \n");
firstInit = getchar();
lastInit = getchar();
```

Вы можете предположить, что если пользователь ввел инициалы *GT*, то в переменную `firstInit` будет записана буква `G`, а в `lastInit` — `T`, однако это не так. Выполнение первого вызова функции `getchar()` не завершается до тех пор, пока пользователь не нажмет клавишу **Enter**, так как введенная буква `G` была записана в буфер. Только после нажатия клавиши **Enter** буква `G` покидает буфер и передается программе, но **Enter** *все еще* хранится в буфере. Таким образом, второй вызов функции `getchar()` посылает **Enter** (а по факту — управляющую последовательность `\n`, представляющую нажатие клавиши **Enter**) в переменную `lastInit`. Буква `T` остается во власти следующего вызова функции `getchar()`, если таковой предусмотрен.

СОВЕТ

Один из способов исправления сложившейся ситуации заключается во введении еще одного, дополнительного, вызова функции `getchar()`, принимающего факт нажатия клавиши **Enter**, но не производящего с ним никаких дальнейших действий.

Далее приведено решение, позволяющее обойти проблему получения инициалов пользователя:

```
printf("Введите первые буквы имени и фамилии: \n");
firstInit = getchar();
n1 = getchar();
lastInit = getchar();
n1 = getchar();
```

При выполнении приведенного кода пользователю потребуется нажимать клавишу **Enter** по введении каждого инициала.

Над переменной `n1` никаких действий производить не нужно, так как она была введена только для хранения фактов нажатия пользователем клавиши **Enter**. Вообще, сам факт нажатия пользователем клавиши **Enter** можно даже не сохранять в отдельной переменной. Следующий код, показывает, как это осуществить.

```
printf("Введите первые буквы имени и фамилии: \n");
firstInit = getchar();
getchar(); //Сброс символа новой строки
lastInit = getchar();
getchar(); //Сброс символа новой строки
```

Некоторые компиляторы языка C выводят предупреждение, если вы используете функцию `getchar()` саму по себе, впрочем, если вы используете вызовы данной функции просто для сброса информации о нажатии пользователем клавиши **Enter**, вы можете не обращать внимания на подобные предупреждения.

Вы также можете потребовать нажатия клавиши **Enter** после введения пользователем двух инициалов, как показано далее:

```
printf("Введите первые буквы имени и фамилии: \n");
firstInit = getchar();
lastInit = getchar();
getchar();
```

Если пользователь введет инициалы *GP*, а затем нажмет клавишу **Enter**, буква `G` будет записана в переменную `firstInit`, а буква `P` — в `lastInit`.

Чуть быстрее: функция *getch()*

Функция для ввода символов `getch()` позволяет избавиться от лишней записи о нажатии клавиши **Enter**, остающейся в памяти после использования функции `getchar()`. Функция `getch()` является *безбуферной*, это, в свою очередь, означает, что функция `getch()` сразу же принимает нажатие любой клавиши без ожидания нажатия пользователем клавиши **Enter**. Недостатком функции `getch()` является невозможность коррекции неправильного ввода нажатием клавиши **Backspace**. Так, с помощью функции `getchar()` пользователь мог бы нажать на клавишу **Backspace**, если он по ошибке напечатал букву `B` вместо `D`. Клавиша **Backspace** удалила бы из буфера букву `B` — и функции `getchar()` досталась бы буква `D`, переданная после нажатия клавиши **Enter**. Так как функция `getch()` *не* буферизирует ввод, то у пользователя нет возможности нажать клавишу **Backspace**. Следующий код принимает два символа без нажатия клавиши **Enter** для подтверждения ввода каждого из символов:

```
printf("Введите первые буквы имени и фамилии: \n");
firstInit = getch();
lastInit = getch();
```

Функция `getch()` немного быстрее функции `getchar()`, так как данная функция не ожидает нажатия клавиши **Enter** перед тем, как воспринять нажатую пользователем клавишу и продолжить работу. Таким образом пропадает необходимость использовании отдельной функции `getch()`, удаляющей информацию о вводе управляющей последовательности `\n`, что было необходимо при работе с функцией `getchar()`.

ВНИМАНИЕ

В отличие от функции `getchar()`, функция `getch()` не выводит на экран вводимые пользователем данные, поэтому после каждого использования функции `getch()` необходимо использовать зеркальную функцию `putch()`, если вы хотите, чтобы пользователь видел на экране введенные им символы. Для отображения введенных инициалов вы могли бы поступить следующим образом:

```
printf("Введите первые буквы имени и фамилии: \n");
firstInit = getch();
putch(firstInit);
```

```
lastInit = getch();
putch(lastInit);
```

В следующей главе мы поговорим еще о нескольких встроенных функциях, в том числе о двух функциях, позволяющих легко вводить и выводить строки, причем этот процесс не сложнее, чем ввод и вывод символов с помощью функций, которые мы обсудили в этой главе.

Абсолютный минимум

Целью данной главы было рассказать о нескольких дополнительных функциях ввода и вывода. Функции, представленные в этой главе — это функции ввода-вывода (I/O) символов. В отличие от функций `scanf()` и `printf()` функции `getchar()`, `getch()`, `putchar()` и `putch()` могут вводить или выводить только по одному символу за раз. Далее приведены основные концепции текста главы:

- Для ввода и вывода отдельных символов воспользуйтесь функциями `getchar()` и `putchar()`.
- Для удаления из памяти информации о нажатии клавиши Enter воспользуйтесь функцией `getchar()`, употребленной самой по себе. Вы также можете создать цикл, который бы продолжал повторно вызывать функцию `getchar()` до тех пор, пока возвращаемое функцией значение не равнялось бы `\n`, как показано в одном из примеров главы.
- Используйте функцию `getch()` для получения *небуферизируемых* символов по мере их ввода пользователем с клавиатуры.
- Не используйте функцию ввода-вывода символов с символьным типом данных, вместо этого, воспользуйтесь целочисленным типом `int`.
- Не забудьте выводить на печать вводимые символы с помощью функции `putch()`, если хотите, чтобы вводимые данные отображались на экране сразу по мере их ввода пользователем.

Глава 19
ПОЛУЧАЕМ БОЛЬШЕЕ ОТ СТРОК

В этой главе

- Применение функций, проверяющих символы
- Проверка правильности регистра
- Подключение функций, изменяющих регистр
- Использование строковых функций

В данной главе мы покажем вам несколько способов снятия горы работы с ваших плеч и возложения ее на язык C. В языке C предусмотрено большое количество полезных функций в дополнение к уже изученным нами ранее `strlen()`, `getchar()` и `printf()`. В языке C предусмотрено очень много встроенных функций: гораздо больше, чем места для них в одной главе. Данная глава посвящена самым популярным и самым полезным символьным и строковым функциям. В следующей главе вы более подробно познакомитесь с несколькими числовыми функциями.

Функции, проверяющие символы

В языке C реализовано несколько встроенных *функций, проверяющие символы*. Теперь, когда вы умеете использовать функции `getchar()` и `getch()`, принимающие символы, функции, проверяющие символы, помогут вам определить, введенные символы какого типа принимает ваша программа. Вы можете реализовать программную логику, основанную на операторе ветвления `if` и предполагающую различное развитие событий, в зависимости от результатов проверки введенных пользователем данных.

СОВЕТ

При использовании в своих программах функций, описываемых в своей главе, не забывайте подключать заголовочный файл `ctype.h` в самом начале файла исходного кода.

Функция isalpha() возвращает значение ИСТИНА (что в языке C соответствует 1), если значение в скобках — символ алфавита от a до z (или символ верхнего регистра от A до Z), и ЛОЖЬ (что в языке C соответствует 0), если значение в скобках представляет собой любой другой символ. Рассмотрим следующий оператор if:

```
if (isalpha(inChar))
{
printf("Вы ввели букву\n");
}
```

В языке C также предусмотрена соответствующая функция isdigit(), возвращающая значение ИСТИНА, если символ в кавычках — число от 0 до 9. Следующий оператор if выводит сообщение Число, если переменная inChar содержит число:

```
if (isdigit(inChar))
{
printf("Число\n");
}
```

ПРИМЕЧАНИЕ

Вы поняли, почему эти функции называются функциями, проверяющими символы? Обе функции isalpha() и isdigit() тестируют символьное содержимое — и возвращают результат реляционного теста.

Верен ли регистр?

Функции isupper() и islower() информируют вас о том, содержит ли переменная символ верхнего или нижнего регистра соответственно.

С помощью вы можете избежать написания длинных выражений с оператором if, например таких:

```
if ((inLetter >= 'A') && (inLetter <= 'Z'))
{
printf("Буква верхнего регистра\n");
}
```

Вместо этого, функция `isupper()` заменяет собой операцию логического сравнения:

```
if (isupper(inLetter))
{
printf("Буква верхнего регистра\n");
}
```

СОВЕТ

Функция `islower()` проверяет переменную на содержание символа нижнего регистра точно также, как функция `isupper()` проверяет на содержание символа верхнего.

Возможно, функция `isupper()` может вам понадобиться для проверки того, что пользователь ввел заглавную первую букву имени или фамилии.

Далее приведена маленькая и быстрая программа, принимающая имя пользователя и пароль, а затем, используя функции, описанные в этой главе, проверяет, содержит ли пароль хотя бы один символ верхнего и нижнего регистра, а также хотя бы одну цифру. Если пароль, введенный пользователем, содержит все три типа символов, программа поздравляет пользователя с тем, что он или она придумали пароль достаточно высокой сложности, создающий большое количество вариантов и усложняющий процесс взлома. Если введенный пароль не содержит хотя бы по одному символу всех трех категорий, программа предлагает пользователю придумать более сложный пароль.

```
//  Программа-пример №1 из главы 19
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter19ex1.c
/* Эта программа запрашивает имя пользователя
и пароль. Программа проверяет, содержит ли введенный
пароль символы верхнего, нижнего регистра, а также
хотя бы одну цифру. Если да, то программа поздравляет
пользователя с удачным выбором пароля. Если нет —
программа предлагает придумать пароль, подразумевающий
больше вариантов с целью повышения уровня
безопасности. */
```

```
//файл stdio.h нужен для функций printf() и scanf()
//файл string.h нужен для функции strlen()
//файл ctype.h нужен для функций isdigit, isupper и islower
#include <stdio.h>
#include <string.h>
#include <ctype.h>
main()
{
int i;
int hasUpper, hasLower, hasDigit;
char user[25], password[25];
//Инициализировать все три переменные 0, т.е. ЛОЖЬю
hasUpper = hasLower = hasDigit = 0;
printf("Как Вас зовут? ");
scanf(" %s", user);
printf("Пожалуйста, придумайте пароль: ");
scanf(" %s", password);
//Данный цикл проходит по всем символам пароля
//и проверяет не является ли символ буквой верхнего,
//нижнего регистра или цифрой
for (i = 0; i < strlen(password); i++)
{
if (isdigit(password[i]))
{
hasDigit = 1;
continue;
}
if (isupper(password[i]))
{
hasUpper = 1;
continue;
}
if (islower(password[i]))
{
```

```
hasLower = 1;
continue;
}
}
/*Данный оператор if  будет выполнен только в том
случае, если все переменные ниже равны 1, а равны 1
они могут быть только в том случае, если в пароле
найдены символы всех трех категорий*/
if ((hasDigit) && (hasUpper) && (hasLower))
{
printf("\n\nХорошая работа, %s,\n", user);
printf("Ваш пароль содержит большие, маленькие буквы");
printf("и цифры.");
} else
{
printf("\n\nПридумайте новый пароль, %s,\n", user);
printf("содержащий большие, маленькие буквы");
printf("и цифры.");
}
return(0);
}
```

В наши дни любой человек, создающий пароль, получает целую лекцию о необходимости включения в пароль целого набора букв, чисел, иногда даже символов, и все для того, чтобы усложнить работу хакерам по взлому таких паролей. В программе для проверки того, что пароль содержит символы всех трех категорий, посредством цикличного посимвольного прохождения и тестирования введенного пароля, используются функции, приводимые в этой главе.

Если проверяемый символ соответствует символу одной из категорий, то значение переменной-флага устанавливается равным `1` (ИСТИНА на языке C), после чего выполнение цикла продолжается.

В первых двух случаях, если значение переменной флага (`hasDigit` и `hasUpper`) устанавливается равным `1`, выражение `continue` запускает новую итерацию цикла: если стало достоверно известно, что рассматриваемый символ — цифра, то необходимости в проведении двух

оставшихся проверок нет (ведь, в конце концов, символ не может принадлежать сразу к нескольким категориям, не так ли?), поэтому из соображений эффективности пропуск последующих тестов — не такая плохая идея. Последний блок кода с оператором `if` не нуждается в выражении `continue`, так как данный оператор и так последний в теле цикла.

После проверки всех символов строки пароля оператор `if` проверяет, удовлетворяет ли пароль всем трем условиям. Если удовлетворяет, то программа выводит на экран поздравительное сообщение, если же нет — на экран выводится сообщение иного характера.

СОВЕТ

Некоторые системы в наши дни запрашивают пароль, содержащий хотя бы один небуквенный и один нечисловой символ (например, `$`, `!`, `*`, `&` и так далее). Вы можете усовершенствовать приведенный ранее программный код, добавив проверку на содержание подобных символов с помощью оператора `else` после последней проверки `islower`. Ведь если символ не соответствует первым трем категориям, то он автоматически подпадает под четвертую.

Функции, изменяющие регистр

В языке C предусмотрено две важные изменяющие символы функции (их также принято называть *функциями изменения кодов символов*). В отличие от функций `isupper()` и `islower()`, которые только *тестируют* символьное значение и возвращают результат ИСТИНА или ЛОЖЬ, функции `toupper()` и `tolower()` возвращают переданный им аргумент с измененным регистром. Функция `toupper()` возвращает аргумент, находящийся в скобках в виде символа верхнего регистра, а функция `tolower()`, соответственно, — нижнего.

Следующий участок программного кода выводит сообщение yes или no, в зависимости от введенных пользователем данных. Без функции `toupper()` вам требуется дополнительный тест для выполнения этого плана:

```
if((userInput == 'Y') || (userInput == 'y'))
{ printf("yes\n");  }
else
{ printf("no\n");  }
```

В следующем наборе выражений используется функция `toupper()` для повышения прямолинейности логического теста оператора `if`, проверяющего на ввод символов нижнего регистра:

```
if(toupper((userInput) == 'Y') //нужно проверить
только символ //верхнего регистра
{ printf("yes\n");  }
else
{ printf("no\n");  }
```

Строковые функции

Заголовочный файл `string.h` содержит не только описание функций `strcpy()` и `strlen()`. В этом разделе речь пойдет о функции, позволяющей выполнить слияние двух символьных массивов, содержащих строки текста. Название функции `strcat()` можно расшифровать как *конкатенация строк* (от англ. «string concatenation»). Функция `strcat()` принимает одну строку и прибавляет, то есть подставляет эту строку в конец другой строки. Следующий фрагмент кода демонстрирует работу функции `strcat()`:

```
char first[25] = "Сидоров";
char last[25] = " Анатолий";
strcat(first, last); //Подставить last в конец first
printf("Меня зовут $s\n", first");
```

Так выглядит сообщение, выведенное программой на экран:

```
Меня зовут Сидоров Анатолий
```

Функции `strcat()` требуется два строковых аргумента. Функция `strcat()` подставляет вторую строку в конец первой. В примере пробел между именем и фамилией появляется только потому, что массив `last` инициализирован во второй строке с первым символом пробелом перед первой буквой фамилии.

ВНИМАНИЕ

Вы сами несете ответственность за то, чтобы первый массив был достаточно большим, чтобы уместить *обе* строки. Если вы попытаетесь произвести конкатенацию второй строки с концом первой строки, и вторая строка определена с недостаточным количеством

символов для хранения обеих строк, вы можете получить странные и неожиданные результаты.

Так как второй аргумент функции `strcat()` не изменяется, то при желании вы можете в качестве этого аргумента задать обычную строку символов. Функции `puts()` и `gets()` предоставляют простой способ ввода и печати строк текста. Их описание находится в заголовочном файле `stdio.h`, поэтому при использовании функций `puts()` и `gets()` вам не потребуется включать в свою программу дополнительный заголовочный файл. Функция `puts()` выводит строку текста на экран, а `gets()` — принимает строку текста с клавиатуры. В следующей программе демонстрируется использование функций `puts()` и `gets()`. При просмотре исходного кода данной программы обратите внимание, что для ввода и вывода строк мы не использовали ни функцию `printf()`, ни `scanf()`.

```
//  Программа-пример №2 из главы 19
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter19ex2.c
/* Эта программа запрашивает ввод название города
проживания и двухбуквенное сокращение соответствующее
штату. После этого в программе используется функция
для создания новой строки с названием города и штата,
а затем выводит эту строку на экран.*/
//файл stdio.h нужен для функций puts() и gets()
//файл string.h нужен для функции strcat()
#include <string.h>
#include <stdio.h>
main()
{
char city[15];
//2 символа для аббревиатуры штата и 1 для нуль-символа
char st[3];
char fullLocation[18] = "";
puts("В каком городе вы живете? ");
gets(city);
puts("В каком штате вы живете? (2-х букв. аббревиатура)");
gets(st);
```

```
/* Конкатенация строк */
strcat(fullLocation, city);
strcat(fullLocation, ", "); //Вставка запятой и пробела
//между городом
strcat(fullLocation, st); //и аббревиатурой штата
puts("\nВы живете в ");
puts(fullLocation);
return(0);
}
```

СОВЕТ

Функцию `strcat()` нужно использовать трижды: один раз для добавления города, один раз — для добавления запятой и еще один раз — для добавления аббревиатуры штата после названия города.

Далее приведен результат тестового запуска программы.

```
В каком городе вы живете?
Gas City
В каком штате вы живете? (2-х букв. аббревиатура)
IN
Вы живете в
Gas City, IN
```

Функция `puts()` автоматически вставляет разрыв строки в конце всех распечатываемых строк. Если вы не хотите вставить дополнительную пустую строку, управляющую последовательность `\n` использовать не нужно.

СОВЕТ

Функция `gets()` автоматически конвертирует нажатие клавши **Enter** в нуль-символ для гарантии того, что полученные с клавиатуры данные будут представлять собой завершенную нуль-символом строку, а не массивом отдельных символов.

Одна из основных причин предпочтения использования функции `gets()` вместо `scanf()` заключается в том, что вы можете запрашивать

пользователя вводить строки, которые уже содержат пробелы, например полное имя (фамилия, имя и отчество). Ввод названия города как в примере, Gas City, с помощью функции `scanf()` вызвал бы проблемы с обработкой данных. В этом ценность функции `gets()`. Поэтому, если бы мы вернулись к примеру с вводом названия любимого фильма из главы 15 и заменили бы функцию `scanf()` на `gets()`, вы могли бы позволить пользователю вводить названия фильмов, состоящие более чем из одного слова.

Абсолютный минимум

Целью данной главы было продемонстрировать несколько встроенных символьных и строковых функций, которые помогут вам проверять и изменять строки текста. Строковые функции, продемонстрированные в этой главе, одинаково хорошо работают как со строками литералов, так и с массивами. Символьные функции тестируют символы на принадлежность к буквам или цифрам, а также преобразуют символы верхнего и нижнего регистра в противоположные. Далее приведены основные концепции текста данной главы.

- Встроенные функции языка C по проверке символов и изменения регистров облегчают работу программе по определению регистра символьных данных.
- Используйте функцию `gets()` для ввода и функцию `puts()` для вывода строк.
- Используйте функцию `gets()` для ввода строк, содержащих пробелы. Помните, что функция `scanf()` не принимает пробелы.
- Для объединения двух строк в одну используйте функцию `strcat()`.
- Не объединяйте две строки с помощью функции `strcat()`, если не уверены, что первый символьный массив может хранить обе строки после их слияния.
- Не используйте управляющую последовательность `\n` с функцией `puts()`, кроме случаев, когда вам необходимо создать дополнительную пустую строку. Функция `puts()` автоматически добавляет разрыв строки в конец распечатываемых строк.

Глава 20

ВЫСШАЯ МАТЕМАТИКА (ДЛЯ КОМПЬЮТЕРА, НЕ ДЛЯ ВАС!)

В этой главе

- Практикум по математике
- Больше преобразований
- Погружаемся в тригонометрию и другие сложные темы
- Становимся непредсказуемыми

Данная глава расширяет ваши познания в области встроенных функций. В этой главе мы поговорим о математических функциях.

В языке С реализованы механизмы, помогающие проводить математические расчеты, которые не могут быть произведены только за счет математических операторов. Встроенные функции языка С предоставляют процедуры, которые вам не придется создавать самостоятельно.

Большинство встроенных математических функций языка С являются узкоспециальными: это не значит, что их сложно использовать, но цель применения этих функций может заключаться в непростых расчетах. Возможно, вы не найдете полезными многие из функций, описываемые в этой главе, за исключением, тех случаев, когда вам нужно воспользоваться тригонометрическими функциями или функциями высшей математики.

СОВЕТ

Некоторые люди имеют многолетний опыт программирования на языке С, при этом они не испытывают потребности в большинстве этих функций. Вам все же следует прочитать эту главу, чтобы получить общую информацию о том, что можно достичь с помощью языка С, и знать, какие инструменты есть у вас под рукой, если однажды придется воспользоваться этими мощными функциями.

Практикум по математике

Для работы всех описываемых в этой главе функций нужен заголовочный файл math.h. Не забывайте подключать файл math.h вместе с файлом stdio.h, если вам нужно воспользоваться какой-либо математической функцией. Первые несколько математических функций, скорее, не столько математические, сколько арифметические, или числовые. Эти функции нужны для преобразования одного числа из другого.

Функции floor() и ceil() можно назвать, соответственно, функцией «пола» и «потолка»*. Эти функции «снижают» или «возносят» нецелые числа к ближайшему целому значению. Например, если вам нужно подсчитать, сколько долларовых банкнот сдано в сдаче (а сдача состоит не только из долларов, но и из центов), то к сумме могла бы быть применена функция floor(). Следующий пример демонстрирует данные пример:

```
change = amtPaid - cost; //Все значения дробные
dollars = floor(change);
printf("Сдача включает %f долларовых банкнот.\n",
dollars);
```

ВНИМАНИЕ

Несмотря на то, что функции ceil() и floor() преобразуют передаваемые аргументы в целые числа, обе функции возвращают значения с плавающей точкой. Именно поэтому мы вывели значение переменной dollars с помощью кода преобразования %f.

Функция ceil(), будучи противоположностью функции floor(), преобразует число к ближайшему целому по возрастанию. Обе функции ceil() и floor() также работают и с отрицательными числами, как показано ниже:

```
lowVal1 = floor(18.5); //Сохраняет 18.0
lowVal2 = floor(-18.5); // Сохраняет -19.0
hiVal1 = ceil(18.5); // Сохраняет 19.0
hiVal2 = ceil(-18.5); // Сохраняет -18.0
```

* От английских слов *floor* и *ceiling* — «пол» и «потолок». — *Примеч. пер.*

ПРИМЕЧАНИЕ

Результаты с отрицательными числами становятся понятнее, если подумать о направлении увеличения отрицательных чисел. Ближайшее целое число, которое *меньше* –18.5 — это –19. Ближайшее целое число, которое *больше* –18.5 — это –18.

Как видите, эти функции не так уж и плохи, кроме того, в момент, когда вы в них будете нуждаться, они окажутся очень полезными.

Еще несколько преобразований

Следующие две числовые функции также преобразуют числовые значения. Функция `fabs()` возвращает *абсолютное значение* числа с плавающей точкой. Впервые услышав об абсолютном значении числа, вам может показаться, что оно никогда вам не понадобится. Абсолютное значение числа, положительное или отрицательное, всегда есть положительное число. Оба вызова функции `printf()` выводят на экран число `25`:

```
printf("Абсолютное значение 25.0: %.0f.\n",
fabs(25.0));
printf("Абсолютное значение -25.0: %.0f.\n", fabs(-
25.0));
```

ПРИМЕЧАНИЕ

Результат — число с плавающей точкой выводится без дробной части благодаря использованию кода преобразования `%f` со спецификатором `.0`.

Абсолютные значения могут быть полезны при подсчете разницы в возрасте, весе и расстояниях. Например, разница в возрасте — всегда положительное число, независимо от того, в каком порядке вы совершаете вычитание.

Две дополнительные математические функции также могут оказаться полезными, даже если вы не занимаетесь серьезным научным или математическим программированием. Функция `pow()` возводит число в степень, а функция `sqrt()` возвращает квадратный корень из числа.

СОВЕТ

Вычислить квадратный корень из отрицательного числа невозможно. Функция `fabs()` позволяет вам быть уверенным в том, что вы не будете пытаться вычислить квадратный корень из отрицательного числа, преобразовав его сначала в положительное, а уже затем вычислив квадратный корень.

Возможно, следующий рисунок освежит в вашей памяти уроки алгебры в старшей школе. На рис. 20.1 показаны известные математические символы, использующиеся для записи функций `pow()` и `sqrt()`.

Если программист на C делает так:	то математик сделает так:
`x = pow(4, 6);`	$x = 4^{6\times6\times6\times6\times6\times6}$
`x = sqrt(value);`	$x = \sqrt{value}$

Рис. 20.1. Математические символы для функций `pow()` и `sqrt()`

Следующий программный код выводит на печать значение числа 10, возведенного в третью степень, а также квадратный корень из 64:

```
printf("10 в третьей степени: %.0f. \n", pow(10.0, 3.0));
printf("Квадратный корень из 64: %.0f. \n", sqrt(64));
```

Далее приведен экранный вывод функций `printf()`:

```
10 в третьей степени: 1000.
Квадратный корень из 64: 8.
```

Погружаемся в тригонометрию и другие сложные темы

Только небольшому количеству наших читателей могут понадобиться тригонометрические и логарифмические функции. Если вы точно знаете или надеетесь, что вы не окажетесь в кругу этих читателей, можете смело пропустить этот раздел. Тем из вас, кому они нужны уже сейчас, не нужно подробное объяснение, его и не будет.

В табл. 20.1 приведены основные тригонометрические функции. Каждой из этих функций требуется аргумент-радианный показатель.

Табл. 20.1. «Тригонометрические функции языка C

Функция	Описание
`cos(x)`	Возвращает косинус угла x
`sin(x)`	Возвращает синус угла x
`tan(x)`	Возвращает тангенс угла x
`acos(x)`	Возвращает арккосинус угла x
`asin(x)`	Возвращает арксинус угла x
`atan(x)`	Возвращает арктангенс угла x

Повторимся, что вероятность того, что вам понадобится одна из этих функций, мала, разве что вы не займетесь повторением тригонометрии (или если у вас есть ребенок или родственник, изучающий тригонометрию в школе, и вы хотели бы проверить его домашнюю работу). Однако знание возможностей языка никогда не будет лишним.

СОВЕТ

Если вы хотите задать аргумент в градусах, а не радианах, то вы можете выполнить конвертацию из градусов в радианы, воспользовавшись следующей формулой:

```
radians = degrees * (3.14159 / 180.0);
```

В табл. 20.2 приведены основные логарифмические функции:

Табл. 20.2. Логарифмические функции языка C

Функция	Описание
`exp(x)`	Возвращает значение экспоненты, основания натурального логарифма, возведенного в степень, заданную выражением `x(e^x)`.
`log(x)`	Возвращает натуральный логарифм аргумента x. В математике записывается как *ln(x)*. x должен быть положительным
`log10(x)`	Возвращает логарифм по основанию 10 аргумента x. В математике записывается как *lg(x)*. x должен быть положительным

В следующей программе используются функции, описанные ранее в этой главе:

```
//  Программа-пример №1 из главы 20
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter20ex1.c
```

```
/* Эта программа демонстрирует математические функции
из библиотеки файла math.h, таким образом вы можете
сделать домашнюю работу без единой ошибки благодаря
быстрым и легким программам */
#include <stdio.h>
#include <string.h>
#include <math.h>
main()
{
printf("Пора сделать домашнюю работу!\n");
printf("\nРаздел 1: Квадратные корни\n");
printf("Квадратный корень из 49: %.1f\n", sqrt(49.0));
printf("Квадратный корень из 149: %.1f\n",
sqrt(149.0));
printf("Квадратный корень из .49: %.1f\n", sqrt(.49));
printf("\nРаздел 2: Степени\n");
printf("4 в 3-ей степени: %.1f\n", pow(4.0, 3.0));
printf("7 в 5-ой степени: %.1f\n", pow(7.0, 5.0));
printf("34 в степени 1/2: %.1f\n", pow(34.0, .5));
printf("\nРаздел 3: Тригонометрия\n");
printf("Косинус угла 60 градусов: %.3f\n",
cos((60*(3.14159/180))));
printf("Синус угла 90 градусов: %.3f\n",
sin((90*(3.14159/180))));
printf("Тангенс угла 75 градусов: %.3f\n",
    tan((75*(3.14159/180))));
printf("Арккосинус угла 45 градусов: %.3f\n",
    acos((45*(3.14159/180))));
printf("Арксинус угла 30 градусов: %.3f\n",
    asin((30*(3.14159/180))));
printf("Арктангенс угла 15 градусов: is %.3f\n",
    atan((15*(3.14159/180))));
printf("\nРаздел 4: Логарифмические функции\n");
printf("e в степени 2: %.3f\n", exp(2));
```

```
printf("Натуральный логарифм 5: %.3f\n", log(5));
printf("Десятичный логарифм 5: %.3f\n", log10(5));
return 0;
}
```

Далее приведен результат выполнения программы. Компьютер посчитал все гораздо быстрее, чем вы на бумаге и с карандашом в руках, не так ли?

```
Пора сделать домашнюю работу!
Раздел 1: Квадратные корни
Квадратный корень из 49: 7.0
Квадратный корень из 149: 12.2
Квадратный корень из .49: 0.39
Раздел 2: Степени
4 в 3-й степени: 64.0
7 в 5-й степени: 16807.0
34 в степени 1/2: 5.8
Раздел 3: Тригонометрия
Косинус угла 60 градусов: 0.500
Синус угла 90 градусов: 1.000
Тангенс угла 75 градусов: 3.732
Арккосинус угла 45 градусов: 0.667
Арксинус угла 30 градусов: 0.551
Арктангенс угла 15 градусов: 0.256
Раздел 4: Логарифмические функции
е в степени 2: 7.389
Натуральный логарифм 5: 1.609
Десятичный логарифм 5: 0.699
```

Становимся непредсказуемыми

Для игр и различных программ-симуляторов вам очень часто понадобится генерировать случайные числа. Функция языка С `rand()` предназначена как раз для этого. Данная функция возвращает случай-

ное число в диапазоне от 0 до 32767. Для использования функции rand() требуется подключить заголовочный файл stdlib.h (*стандартная библиотека*). Для уменьшения диапазона случайных чисел вы можете воспользоваться оператором деления по модулю %. Следующее выражение записывает в переменную dice случайное число в диапазоне от 1 до 6:

```
dice = (rand() % 5) + 1; /*От 1 до 6 */
```

ПРИМЕЧАНИЕ

Так как игральная кость может иметь значение от 1 до 6, оператор деления по модулю возвращает целочисленный остаток от деления (от 0 до 5), после чего к нему прибавляется 1 для задания значения игральной кости.

Если вы хотите получить *действительно* случайное число, вам нужно сделать кое-что сумасшедшее.

Присвоить генератору случайных чисел *семя* означает дать генератору начальную базу, исходя из которой функция rand() будет отсчитывать случайные числа. Для задания семени воспользуйтесь функцией srand(). Если вы не хотите, чтобы программа при каждом запуске генерировала один и тот же набор случайных чисел, число в скобках функции srand() должно быть всегда разным.

Трюк передачи функции srand() разных значений при каждом запуске программы заключается в передаче в качестве аргумента точного времени дня, причем точность доходит до сотых секунды. Таким образом, сначала с помощью объявления time_t объявите переменную для хранения значения времени, затем передайте адрес этой переменной (с помощью символа & перед именем переменной) функции srand().

ПРИМЕЧАНИЕ

Возможно, вам потребуется, чтобы программа всегда генерировала новый набор случайных чисел при каждом запуске. Такая рандомизация нужна при создании игр, однако многие симуляторы и научные исследования подразумевают повторение одних и тех же случайных чисел. Функция rand() предоставляет как раз такой функционал, если вы не задаете семя генератора случайных чисел.

Прежде чем задавать время дня в качестве семени генератора случайных чисел, подключите к программе заголовочный файл time.h.

Философия здесь следующая: если вы прописываете в программе два странных выражения, функция rand() всегда будет генерировать случайные числа и создавать разный набор случайных чисел при каждом запуске программы.

Как всегда, наилучший способ как прийти к пониманию подобных функций — это увидеть пример их использования. В следующем примере для симуляции броска двух игральных костей и вывода результата на экран используется функция rand(). Затем пользователь должен решить, будет ли результат следующего броска больше, меньше или равным результату предыдущего броска:

```
//  Программа-пример №2 из главы 20
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter20ex2.c
/* Эта программа демонстрирует математические функции
из библиотеки файла math.h, таким образом, вы можете
сделать домашнюю работу без единой ошибки благодаря
быстрым и легким программам */
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <ctype.h>
main()
{
int dice1, dice2;
int total1, total2;
time_t t;
char ans;
//Запомните, что данная функция нужна, чтобы
//генерируемые числа были разными
srand(time(&t)) ;
//Вы получите число от 0 до 5, поэтому +1
//задаст диапазон числа от 1 до 6
dice1 = (rand() % 5) + 1;
```

```
dice2 = (rand() % 5) + 1;
total1 = dice1 + dice2;
printf("Первый бросок игральных костей: %d и %d, ",
dice1, dice2);
printf("с общей суммой %d.\n\n\n", total1);
do {
puts("Как вы думаете, сумма следующего броска будет ");
puts("(H)Больше, (L)Меньше или (S)Равна?\n");
puts("Введите H, L, или S в соответствии с вашим мнением.");
scanf(" %c", &ans);
ans = toupper(ans);
} while ((ans != 'H') && (ans != 'L') && (ans !=
'S'));
// Второй бросок и получение второй суммы
dice1 = (rand() % 5) + 1;
dice2 = (rand() % 5) + 1;
total2 = dice1 + dice2;
// Отобразить результаты второго броска
printf("\nПервый бросок игральных костей: %d и %d, ",
dice1, dice2);
printf("с общей суммой %d.\n\n", total2);
// Теперь сопоставим догадку пользователя
с результатом
// второго броска и сообщим, прав пользователь или нет
if (ans == 'L')
{
if (total2 < total1)
{
printf("Хорошая работа! Вы оказались правы!\n");
printf("%d меньше, чем %d\n", total2, total1);
}
else
{
printf("Извините! %d не меньше %d\n\n", total2, total1);
}
```

```
}
else if (ans == 'H')
{
if (total2 > total1)
{
printf("Хорошая работа! Вы оказались правы!\n");
printf("%d больше, чем %d\n", total2, total1);
}
else
{
printf("Извините! %d не больше, чем %d\n\n", total2,
total1);
}
}
else if (ans == 'S')
{
if (total2 == total1)
{
printf("Хорошая работа! Вы оказались правы!\n");
printf("%d равно %d\n\n", total2, total1);
}
else
{
printf("Извините! %d не равно %d\n\n", total2,
total1);
}
}
return(0);
}
```

Ну что ж, неплохо! Изучив меньше двух третей этой книги, вы можете назвать себя программистом-создателем компьютерных игр! Данная программа — это простая игра в догадки, в которой пользователи должны угадать, как будет соотноситься сумма значений игральных костей второго броска с суммой первого. Программа дает пользователям только один шанс и завершается после сопоставления результатов и со-

общения пользователю степени его правоты. Однако простой цикл `do...while`, заключающий в себя всю программу, мог бы изменить эту ситуацию таким образом, что пользователь будет продолжать играть столько, сколько ему захочется, прежде чем решит выйти из программы. Почему бы вам не попробовать добавить в программу такой цикл?

Абсолютный минимум

Целью данной главы было рассказать о встроенных математических функциях, призванных облегчить обработку числовых данных. В языке С предусмотрен богатый ассортимент целочисленных функций, функций числовых преобразований, функций для работы с датой и временем, а также функций, генерирующих случайные числа.

На данном этапе совсем не обязательно досконально разбираться во всех приведенных в этой главе функциях. Вы можете написать сотни программ на языке С, так ни разу и не воспользовавшись многими из этих функций. Тем не менее они есть в языке С, и вы сможете воспользоваться ими, когда понадобится.

- По возможности пользуйтесь встроенными числовыми функциями, чтобы не писать код самостоятельно для выполнения аналогичных операций.
- Многие числовые функции, такие как `floor()`, `ceil()` и `fabs()`, преобразуют одни числа в другие
- Не забывайте предоставить генератору случайных чисел текущее время дня в качестве семени рандомизации, чтобы функция `rand()` генерировала разный набор чисел при каждом запуске программы.
- Не нужно убивать себя мыслью, что вы должны в совершенстве овладеть тригонометрическими и логарифмическими функциями, даже если сейчас они вам не нужны. Многие программисты, работающие на языке С, никогда ими не пользуются.
- Не используйте целочисленные переменные для записи значений, возвращаемых функциями, описанными в этой главе (если вы не приводите типы возвращаемых значений). Все функции из этой главы возвращают значения типа `float` или `double`, даже несмотря на то, что результатом работы некоторых из них, например функции `ceil()`, является возврат целого числа

Глава 21
РАБОТА С МАССИВАМИ

В этой главе

- Повторение массивов
- Запись значений в массивы

Пожалуй, действительно приятной особенностью этой главы является тот факт, что она не посвящена чему-то новому. Вы на протяжении всей книги уже успели поработать с массивами, когда сохраняли строки текста в символьные массивы. Данная глава лишь отшлифовывает понятие массив и демонстрирует вам, что вы можете создавать массивы абсолютно любого типа данных, а не только типа `char`.

Как вы уже знаете, массив символов — это лишь список символов, имеющий имя. Точно так же массив целых чисел — это список целых чисел, имеющих имя, а массив чисел с плавающей точкой — лишь имеющий имя перечень чисел с плавающей точкой. Вместо ссылки на каждый элемент массива по уникальному имени вы ссылаетесь на элемент массива по общему имени массива и выделяете нужный вам элемент его порядковым номером, заключенным в квадратные скобки.

Повторение массивов

Все массивы содержат значения, называемые *элементами*. Массив может содержать *только* элементы одного типа. Иными словами, вы не можете создать массив, который содержал бы в себе значения типа `float`, символьные и целочисленные значения.

Массивы объявляются почти также, как и переменные — не массивы. Для определения обычной переменной, все, что вам нужно сделать — указать тип данных и имя:

```
int i; /* Объявление переменной — не массива */
```

Чтобы объявить массив, вы должны в объявление добавить квадрантные скобки ([]) после имени и указать максимальное количество элементов, которые вы сможете сохранить в данном массиве:

```
int i[25]; /* Объявление массива */
```

Если вы хотите инициализировать символьный массив какой-либо начальной строкой, вы можете сделать это следующим образом:

```
char name[6] = "Italy"; /* Оставьте место для нуль-
символа! */
```

ВНИМАНИЕ

После объявления массива определенного размера не пытайтесь сохранить в нем элементов больше, чем было позволено размером массива изначально. После объявления массива `name`, как только что было сделано, функция `strcpy()` *позволяет* сохранить в этом массиве строку текста, длиннее, чем название страны `Italy`, но результат этого действия был бы катастрофичным, так как более длинная строка может случайно перезаписать другие данные, хранящиеся в оперативной памяти. Если случится так, что другая переменная будет объявлена сразу после массива `name`, то данные этой переменной будут перезаписаны, если вы попытаетесь сохранить слишком длинную строку в массиве `name`.

Если изначально массив должен быть больше, чем значение, присваиваемое ему при инициализации, то при объявлении массива вы можете указать больший размер, например следующим образом:

```
char name[80] = "Italy"; /* Остается много свободного
места */
```

Если вы поступите таким образом, то в массиве `name` будет достаточно места для сохранения строки, куда длиннее, чем слово `Italy`. Например, вы, возможно, захотите воспользоваться функцией `gets()` для получения строки текста от пользователя, которая, опять же, запросто может оказаться куда длиннее слова `Italy`.

Создавайте массивы большого размера, чтобы они могли хранить большее количество значений, однако не следует слишком усердствовать: не создавайте массивы размером больше, чем, вы считаете, может понадобиться в будущем. Массивы могут поглощать большое количе-

ство памяти, и чем больше элементов вы резервируете, тем меньше памяти остается для работы вашей программы и хранения переменных.

Вы можете инициализировать элементы массива один за другим, для этого при объявлении массива заключите значения элементов массива в фигурные скобки после имени массива и знака равно. Например, следующее выражение одновременно объявляет целочисленный массив *и* инициализирует пять его элементов:

```
int vals[5] = {10, 40, 70, 90, 120};
```

Для наглядности рис. 21.1 демонстрирует то, как выглядит массив `vals` в памяти после объявления. В конце массива нет нуль-символа, так как нуль-символ завершает только строки, хранящиеся в символьных массивах.

Массив vals

10	vals[0]
40	vals[1]
70	vals[2]
90	vals[3]
120	vals[4]

Рис. 21.1. После инициализации массива `vals`

ПРИМЕЧАНИЕ

Нумерация индексов всех массивов в языке C начинается с 0.

Следующие два выражения объявляют и инициализируют два массива: массив чисел с плавающей точкой и массив чисел с плавающей точкой двойной точности. Так как язык C отличается достаточно свободным синтаксисом выражений, вы можете продолжать инициализацию элементов массива на нескольких строчках, как это сделано ниже на примере массива `annualSal`:

```
float money[10] = {6.23, 2.45, 8.01, 2.87, 6.41};
double annualSal[6] = {43565.78, 75674.23, 90001.34,
            10923.45, 39845.82};
```

Вы также можете объявить символьный массив и одновременно инициализировать содержащиеся в нем отдельные символы:

```
char grades[5] = {'A', 'B', 'C', 'D', 'F'};
```

Так как последний элемент массива не содержит нуль-символ, говорят, что массив grades содержит отдельные символы, а не строку текста. Если бы последний элемент был инициализирован как '/0' (управляющая последовательность, представляющая на письме нуль-символ), то вы могли бы обрабатывать массив grades как строку и выводить его на печать с помощью функции puts() или printf() и кода преобразования %s.

Следующее объявление записывает строку текста в массив name:

```
char italCity[7] = {'V', 'e', 'r', 'o', 'n', 'a'\0};
```

Однако следует признать, что массив, содержащий строку символов, было бы проще инициализировать следующим образом:

```
char italCity[7] = "Verona"; /* автоматический нуль-
символ */
```

Впрочем, вернемся к числовым массивам: они являются основной темой этой главы. Присутствует ли нуль-символ в конце следующего массива с именем nums?

```
int nums[4] = {5, 1, 3, 0};
```

Правильный ответ — *нет*, в конце массива nums нуль-символа нет! Будьте внимательны: массив nums не является символьным, и в нем не хранится строка текста. Нуль в конце массива — обычный арифметический нуль. Битовый шаблон (это модный компьютерный термин для «внутреннего представления данных») совпадает с шаблоном нуль-символа, но вы никогда не сможете обрабатывать массив nums так, как если бы в этом массиве хранилась строка текста, поскольку массив nums объявлен числовым.

ВНИМАНИЕ

Всегда указывайте количество элементов при объявлении массива. Но у этого правила есть одно исключение: при присвоении изначального значения или набора значений *в момент объявления массива* вы можете оставить скобки пустыми:

```
int ages[5] = {5, 17, 65, 40, 92}; //Правильно
int ages[]; //Неправильно
int ages[] = {5, 17, 65, 40, 92}; //Правильно
```

ПРИМЕЧАНИЕ

Функция `sizeof()` возвращает количество байтов, которое вы *зарезервировали*, а не количество элементов, в которых вы записали какое-либо значение. Например, если на вашем компьютере числа с плавающей точкой занимают 4 байта, то 8-элементный массив чисел с плавающей точкой занял бы в общем 32 байта памяти, и 32 — это как раз то значение, какое вернет функция `sizeof()`, примененная для измерения размера объявленного массива.

Если вы хотите обнулить все элементы массива, вы можете осуществить это с помощью особого сокращения, предусмотренного в языке C:

```
float amount[100] = {0.0}; /* Обнуление всего массива */
```

Если вы не инициализируете массив, компилятор C не сделает это за вас: до тех пор, пока вы не запишите в массив какие-либо значения, вы не можете знать, что хранится в массиве. Исключение из этого правила заключается в том, что большинство компиляторов языка C обнуляют все элементы массива, если вы инициализируете хотя бы один элемент при объявлении массива. Это правило сработало для массива `amount`, так как одно значение было записано в первый элемент данного массива, а остальные элементы были автоматически заполнены нулями. (Даже если бы первый элемент был инициализирован значением `123.45`, компилятор все равно заполнил бы остальные элементы нулями.)

Запись значений в массивы

Вы не всегда можете знать содержимое массива в момент его объявления. Зачастую значения элементов массива берутся из файла, хранящегося на диске, вычислений или пользовательского ввода. Благодаря встроенной функции языка C `strcpy()` вы с легкостью можете заполнить символьный массив строкой текста. Однако для массовой записи в массив целых чисел или чисел с плавающей точкой функции, подобной `strcpy()`, не существует.

Следующий программный код определяет целочисленный массив и запрашивает у пользователя ввод значений, сохраняемых в этом массиве. В отличие от обычных переменных, отличающихся уникальными именами, с элементами массива гораздо удобнее работать, потому что вы можете создать цикл, отсчитывающий номера элементов, как сделано в следующем примере:

```
int ages[3];
for (i = 0; i < 3; i++)
{
  printf("Введите возраст ребенка №%d ", i+1);
  scanf(" %d", &ages[i]); //Получает от пользователя
следующий возраст
}
```

Теперь давайте воспользуемся простой программой, сочетающей оба метода ввода данных в массив. Данная программа следит за количеством очков, полученных игроком в каждом из 10 баскетбольных матчей. Первые шесть очков вводятся в момент инициализации массива, после чего программа просит пользователя ввести очки за матчи 7—10. После ввода всех данных программа с помощью цикла проходит по всем 10 значениям и вычисляет среднее количество очков за игру:

```
//  Программа-пример №1 из главы 21
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter21ex1.c
/* Эта программа создает массив для хранения очков
пользователя, полученных в ходе 10 баскетбольных матчей.
Очки, полученные за первые 6 игр уже находятся в программе,
а очки за последние 4 игры вводятся пользователем. */
#include <stdio.h>
main()
{
int gameScores[10] = {12, 5, 21, 15, 32, 10};
int totalPoints = 0;
int i;
float avg;
//Нужны только очки за последние 4 игры, поэтому цикл
```

```
//пройдет по элементам массива 6-9
for (i=6; i < 10; i++)
{
//Прибавить к единицу к номеру элемента массива, чтобы
//пользователь воспринимал первую игру как игру 1, а не 0
printf("Очки игрока за игру №%d: ", i+1);
scanf(" %d", &gameScores[i]);
}
//Теперь, получив все 10 значений, проходим по ним
//для получения их суммы и вычисления среднего значения.
for (i=0; i<10; i++)
{
totalPoints += gameScore[i];
}
//Для среднего значения используем переменную типа float,
//так как среднее значение, вероятно, окажется между
//двух целых
avg = ((float)totalPoints/10);
printf("\n\nСреднее количество очков игрока: %.1f.\n",
avg);
return 0;
}
```

Далее приведен результат тестового запуска данной программы:

```
Очки игрока за игру №7: 21
Очки игрока за игру №8: 8
Очки игрока за игру №9: 11
Очки игрока за игру №10: 14
Среднее количество очков игрока: 14.9
```

Таким образом, целью создания данной программы было показать два различных способа записи значений в массив. Данная программа несколько обезличена, так что при желании вы могли бы добавить в начало программы строковый массив для хранения имени игрока, впоследствие запросы на ввод очков могли бы также включать имя этого пользователя. В следующих двух главах вы узнаете, как осуществлять

поиск в массиве, а также сортировать данные, хранящиеся в массиве, например, на случай, если вам нужно отсортировать очки пользователя от лучшего к худшему.

ВНИМАНИЕ

Не повторяйте нашей ошибки. При первом запуске данной программы мы получили средний результат, равный 42 000 очков за игру (что, конечно, было бы рекордом для каждого игрока). Почему это произошло? При объявлении переменной `totalPoints` мы не инициализировали ее с начальным значением `0`, и, как мы напоминали вам уже несколько раз по тексту этой книги, но не применили на собственной программе, вы не можете предполагать, что переменная будет изначально пустой или иметь значение `0` только потому, что вы ее объявили.

Абсолютный минимум

Целью данной главы было научить вас способам сохранения данных в виде списков, именуемых *массивами*, т. е. наборами переменных. Все переменные носят одно и то же имя (имя массива), но отличия переменных в массиве (*элементов массива*) заключаются в числовом *индексе*. Первый элемент массива всегда имеет индекс `0`, нумерация остальных идет по возрастанию от индекса первого элемента.

Характерной чертой массива являются квадратные скобки, следующие сразу же за именем массива. При необходимости ссылки на определенный элемент массива индекс этого элемента помещают в квадратные скобки. Основные концепции главы приведены далее.

- Используйте массивы для хранения списков значений одного и того же типа.
- Ссылайтесь на нужный элемент массива по его индексу.
- Если вы хотите «пройти» по всем элементам массива, создайте соответствующий цикл `for`, будь то для инициализации, печати или изменения значений элементов массива.
- Не используйте больше элементов массива, чем объявлено индексов элементов.
- Не используйте массив до момента его инициализации определенными значениями.

Глава 22
ПОИСК В МАССИВАХ

В этой главе

- Заполнение массивов
- Поиск значений в параллельных массивах

Вы купили эту книгу, чтобы безболезненно выучить язык программирования C, это именно то, что происходило до сих пор. (Вы знали, что что-то *происходит*, не так ли?) Тем не менее вы не станете хорошим программистом, если вас не поставить лицом к лицу с задачами по сортировке и поиску значений. Техникам сортировки и поиска значений посвящены целые книги, а следующие две главы представляют вам лишь две простейшие техники. Однако предупреждаем вас заранее, что эта и следующая главы поднимут вопросов больше, чем смогут дать ответов.

Эту и следующую главу вы найдете отличающимися от большинства глав этой книги. Вместо обучения вас новым особенностям языка C эти две главы демонстрируют использование уже изученных вами по мере чтения этой книги элементов языка C. Основной темой обеих глав являются массивы. В этих главах вы встретите практическое применение концепций, изученных вами в главе 21. После того, как они усилят ваше понимание массивов, глава 25 расскажет вам об альтернативе массивам, предусмотренной в языке C, которая иногда может оказаться полезной.

Заполнение массивов

Как уже было упомянуто в главе 21, в программах может применяться несколько способов заполнения массивов. Значения некоторых массивов, содержащих, к примеру, количество дней в каждом из 12 месяцев года, исторические данные измерения температуры воздуха, результаты коммерческой деятельности за предыдущий год известны заранее. Вы можете инициализировать такие массивы имеющимися данными в момент объявления этих массивов либо с помощью операции присваивания.

Вы также можете заполнять массивы данными, вводимыми пользователями. Например программа, отслеживающая выполнение заказа клиента, получает необходимые данные только тогда, когда клиент делает заказ. Аналогично этому лаборатория узнает результаты тестов только после того, как ученые соберут полученные результаты.

Другие данные приходят в программу из файлов, хранящихся на диске. Информация о клиентах, данные о складских остатках и школьный табель успеваемости — это примеры очень объемных данных, которые сложно вводить вручную при каждом запуске программы.

В реальной жизни ваши программы могут и будут заполнять массивы, комбинируя следующие три метода:

- Присваивание
- Ввод данных пользователем
- Файлы, хранящиеся на диске

Одной из задач этой книги является упрощение задач-примеров, поэтому до того момента, как вы изучите виды файлов, хранящихся на диске, вы будете работать с программами, в которых массивы заполняются с помощью операции присваивания или, возможно, ввода небольшого количества данных пользователем с клавиатуры (а пользователем будете *вы*!).

ПРИМЕЧАНИЕ

На данном этапе вам важнее сконцентрировать свое внимание на том, что происходит с массивами после заполнения их значениями. Одной из важнейших задач является *поиск* введенных в массив значений.

Находчики, хранители

Представьте следующую ситуацию: ваша программа содержит массив, в который записаны идентификационные номера клиентов, а также массив, содержащий такое же количество балансов клиентов. Такие массивы зачастую называют *параллельными массивами*, так как они находятся в синхронизированном состоянии, иными словами, элемент №14 содержит идентификационный номер клиента, имеющий задолженность, которую можно найти в элементе №14 массива с балансами.

Программа обработки балансов клиентов может заполнить оба массива данными с диска при первом запуске. По мере размещения клиентом нового заказа задача вашей программы заключается в поиске данных о балансе данного клиента и предотвращении размещения заказа, если клиент уже задолжал вам 100 долларов (халявщик!).

В общих чертах задачу программы можно описать следующим образом:

1. Запросить идентификационный номер клиента (ключ).
2. Совершить поиск баланса клиента в массиве, который соответствовал бы введенному ключу.
3. Проинформировать вас о том, что клиент уже должен вам 100 долларов.

Следующая программа как раз выполняет три вышеуказанные действия. На самом деле, в программе хранится список только из 10 клиентов, так как вы еще не готовы взяться за дисковый ввод (но вы уже почти к этому пришли!). Программа инициализирует массивы при их объявлении, поэтому хранение только 10 пар элементов (массив идентификационных номеров клиентов и соответствующий массив балансов) позволяет оставить объявление массивов простым.

Изучите эту программу перед вводом и запуском, посмотрите, сможете ли вы разобраться в сути программы, просто просмотрев программный код и комментарии к нему. Объяснение работы программы находится сразу после листинга.

```
//  Программа-пример №1 из главы 22
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter22ex1.c
/* Эта программа принимает вводимый пользователем
идентификационный номер клиента и сравнивает его
с номерами в базе данных. Если введенный номер
присутствует в базе, то программа использует данный
элемент массива для проверки текущего баланса
пользователя и выводит предупреждение, если баланс
пользователя больше 100 долларов */
#include <stdio.h>
main()
{
```

```
int ctr; //Счетчик цикла
int idSearch; //Искомый клиент (ключ)
int found = 0; //Будет равен 1 (ИСТИНА) если клиент найден
//Определение 10 элементов двух параллельных массивов
int custID[10] = {313, 453, 502, 101, 892,
475, 792, 912, 343, 633};
int custBal[10] = {0.00, 45.43, 71.23, 301.56, 9.08,
192.41, 389.00, 229.67, 18.31, 59.54};
/*Взаимодействие с пользователем, ищущим баланс. */
printf("\n\n*** Проверка баланса клиента ***\n");
printf("Введите искомый номер клиента: ");
scanf(" %d", &idSearch);
/*Поиск и установление наличия номера клиента
в массиве*/
for (ctr=0; ctr<10; ctr++);
{
if (idSearch == custID[ctr])
{
found = 1;
break;
}
}
if (found)
{
if (custBal[ctr] > 100.00)
{
printf("\n**Баланс клиента: $%.2f.\n", custBal[ctr]);
printf("  Кредит недоступен.\n");
}
else
{
printf("\n**У клиента хороший кредитный баланс!\n");
}
}
```

```
else
{
printf("**Вы ввели неверный ID клиента.");
printf("\n  Введенный ID %3d в списке не найден.\n",
idSearch);
}
return(0);
}
```

Поиск клиента с помощью программы имеет три возможных исхода:

- Баланс клиента меньше 100 долларов.
- Баланс клиента слишком велик (больше 100 долларов).
- Идентификационный номер клиента отсутствует в списке.

Далее приведены три тестовых запуска программы, демонстрирующие каждую из трех возможностей:

```
*** Проверка баланса клиента ***
Введите искомый номер клиента: 313
**У клиента хороший кредитный баланс!
*** Проверка баланса клиента ***
Введите искомый номер клиента: 891
**Вы ввели неверный ID клиента.
  Введенный ID 891 в списке не найден.
*** Проверка баланса клиента ***
Введите искомый номер клиента: 475
**Баланс клиента: $192.41
  Кредит недоступен.
```

В первой части программы объявляются и инициализируются два массива: с идентификационными номерами клиентов, а также соответствующими балансами. Как вы уже знаете, при первичном объявлении массива вы можете использовать оператор присваивания (=) для записи данных в массив.

После вывода на экран заголовка и запроса идентификационного номера клиента программа, с помощью цикла `for`, проходит по параллель-

ным массивам в поисках введенного пользователем идентификационного номера клиента. Если идентификационный номер найден, значение переменной `found` устанавливается в `1` (ИСТИНА) для дальнейшего использования. В противном случае значение переменной `found` остается равным `0` (ЛОЖЬ).

СОВЕТ

Переменные, наподобие переменной `found` из нашего примера, зачастую называются *флагами*, так как данная переменная сигнализирует (как бы, поднимая флаг) остальной программе, был ли найден идентификационный номер клиента.

Цикл `for` может завершиться, так и не найдя искомого клиента. Код, следующий за циклом `for`, не может знать, завершился цикл досрочно благодаря выполнению команды `break` (что означает, что программе удалось найти искомого клиента) или же цикл завершился нормально. Поэтому переменная `found` информирует код, следующий за циклом `for`, был ли найден искомый клиент.

Когда `for` завершен, клиент либо найден, либо не найден. Если клиента удалось найти, то возможны следующие два условия:

- Кредитный баланс этого клиента уже слишком велик.
- Текущий баланс допускает операции в кредит.

Вне зависимости от того, которое из условий истинно, программа информирует пользователя о результате. Если клиента найти не удалось, программа сообщает об этом пользователю и завершается.

Как вам такая программа из *реальной жизни*? Слишком сложная, вы скажете? Тогда просмотрите ее еще один или даже два раза — и вы увидите, что программа выполняет те же самые шаги (хоть и гораздо более детально), что выполнили бы и вы при просмотре списка клиентов вручную.

ПРИМЕЧАНИЕ

Что *на самом деле* важно, так это то, что если бы в базе была тысяча или даже 10000 клиентов и массивы инициализировались данными из файла на диске, тот же самый поисковый программный код также справился бы с задачей! Количество данных не влияет на логику программы (за исключением способа инициализации массива данными).

Далее приведена еще одна программа, показывающая ценность связанных массивов. Возвращаясь к баскетболисту из предыдущей главы, данная программа содержит три массива: один для очков, один для ребаундов (подборов мяча) и еще один для голевых передач. Программа осуществляет поиск среди результатов по очкам, находит игру, в которой игрок заработал наибольшее количество очков, а затем распечатывает общий результат игрока по всем трем категориям в этой конкретной игре:

```
//  Программа-пример №2 из главы 22
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter22ex2.c
/* Эта программа заполняет три массива очками игрока,
количеством подборов и голевых передач, затем проходит
по массиву с очками и находит игру, в которой игрок
получил наибольшее их количество. По получении этой
информации программа распечатывает общий результат
игрока по всем трем категориям в этой конкретной игре*/
#include <stdio.h>
main()
{
int gameScores[10] = {12, 5, 21, 15, 32, 10, 6, 31, 11, 10};
int gameRebounds[10} = {5, 7, 1, 5, 10, 3, 0, 7, 6, 4};
int gameAssists[10] = {2, 9, 4, 3, 6, 1, 11, 6, 9, 10};
int bestGame = 0; //Сравнительная переменная для очков
самой //результативной игры
int gmMark = 0; //Данная переменная отметит самую
//результативную игру
int i;
for (i=0; i<10; i++)
{
//цикл for сравнит результат каждой игры с текущим
//наилучшим результатом, если этот результат
оказывается
//выше, он становится наилучшим и переменная-счетчик
//становится новым значением переменной gmMark
if (gameScore[i] > bestGame)
```

```
{
bestGame = gameScores[i];
gmMark = i;
}
}
//Распечатать результаты поиска самой результативной
игры
//Т.к. индексы массивов начинаются с 0, прибавить 1
к № игры
printf("\n\nОчки игрока в самой результативной
игре:\n");
printf("Самой результативной стала игра №%d\n",
gmMark+1);
printf("Заработано %d очков\n", gameScores[gmMark]);
printf("Подобрано %d мячей\n", gameRebounds[gmMark]);
printf("Осуществлено %d голевых передач\n",
gameAssists[gmMark]);
return 0;
}
```

СОВЕТ

Если вы отслеживаете несколько переменных, связанных с одним и тем же объектом (например, различные показатели игрока за одну игру), более четко все связать воедино вам поможет структура. Вы изучите структуры в главе 27 «Упорядочивание данных с помощью структур».

В обеих задачах-примерах из этой главы используется *последовательный поиск*, так как поиск в массивах (идентификационный номер клиента и gameScores) осуществляется от начала массива до конца, до того, как будет найдено совпадение. По мере увеличения ваших навыков программирования вы узнаете о более совершенных способах поиска данных. В следующей главе вы увидите, как сортировка данных массива позволяет ускорить процесс поиска данных. Вы также ознакомитесь с двумя более совершенными техниками поиска, называемыми *бинарным поиском* и *поиском делением по числам Фибоначчи*, или *поиском Фибоначчи*.

Абсолютный минимум

Целью данной главы была демонстрация метода поиска данных в массиве. Вы узнали, как осуществить поиск данных в массиве с помощью *ключа* — данных, вводимых пользователем. В реальной жизни вам часто потребуется осуществлять поиск данных в параллельных массивах. Один из таких массивов (ключевой массив) содержит данные, среди которых осуществляется поиск. Если поиск завершается успешно, остальные массивы предоставляют необходимые данные — и вы можете оповестить пользователя о результатах поиска. Если поиск окончился неудачей, вы также должны сообщить пользователю об этом. Далее приведены основные концепции данной главы.

- Заполнение массива — лишь первый шаг. После того как массивы заполнены, ваша программа должна взаимодействовать с имеющимися данными.
- До того момента, как вы познакомитесь с более совершенными техниками поиска, пользуйтесь техникой последовательного поиска, так как данная техника — самая простая из изучаемых.
- Не забывайте о том, что соответствие может быть так и не найдено. Всегда предполагайте, что искомое значение может не находиться в списке значений, поэтому всегда предусматривайте код для обработки ненайденных значений

Глава 23
СОРТИРОВКА ПО АЛФАВИТУ И УПОРЯДОЧЕНИЕ ДАННЫХ

В этой главе

- Уборка в доме: сортировка
- Проведение более быстрого поиска

Сортировка — это компьютерный термин, означающий упорядочение списков значений. Вы должны уметь не только найти данные в массиве, но также зачастую и отсортировать их в определенной последовательности. Компьютеры идеально подходят для сортировки и упорядочения данных, а массивы предоставляют эффективные средства хранения сортированных данных.

Не всегда данные хранятся в программах именно в том виде, в каком вы хотели бы их увидеть на экране. Например, абитуриенты не подают заявления в вуз в алфавитном порядке, хотя большинство образовательных учреждений распечатывают списки поступивших именно в алфавитном порядке. Таким образом, после сбора данных о студентах компьютерные программы вуза должны каким-то образом выстроить эти данные в алфавитном порядке по буквам фамилий будущих студентов.

Эта глава посвящена простейшему методу компьютерной сортировки данных: *пузырьковой сортировке*.

Приберемся в доме: сортировка

Если вы хотите упорядочить по алфавиту список букв или имен или же расположить список сумм продаж по *возрастанию* (по возрастанию означает от меньшего к большему, а по *убыванию* — от большего к меньшему), вам потребуется воспользоваться процедурами сортировки. Разумеется, список упорядочиваемых значений будет храниться в массиве, так как значения, хранящиеся в массиве, очень просто переупорядочить с помощью индексов элементов массивов.

Подумайте над тем, как бы вы собирали и упорядочивали колоду брошенных в воздух карт. Вы бы поднимали их с пола одну за одной, смотрели бы, как соотносится только что поднятая карта с картами, уже имеющимися у вас в руках. Зачастую вы будете менять местами поднятые карты. Для сортировки массива применяется аналогичная процедура: зачастую вам необходимо поменять местами значения, уже находящиеся в массиве.

Существует несколько методов компьютерной сортировки значений. В этой главе мы обучим вас методу *пузырьковой сортировки*. Метод пузырьковой сортировки не отличается особой эффективностью в сравнении с другими методами, но он самый простой для понимания. Название «пузырьковая сортировка» подобрано в соответствии с природой данного метода. Во время сортировки значения как бы «всплывают» вверх по списку при каждом прохождении по данным. На рис. 23.1 изображен процесс сортировки пяти чисел по пузырьковому методу.

Следующая программа сортирует список из 10 чисел. Все числа сгенерированы случайно функцией `rand()`. Процедура пузырьковой сортировки несколько более сложна, чем просто вложенный цикл `for`. Внутренний цикл проходит по списку значений, меняя местами любую пару чисел, расположенную не по порядку. Внешний цикл заставляет внутренний цикл повториться несколько раз (по одному разу для каждого элемента цикла).

Бонус, часто встречающийся во многих улучшенных методах пузырьковой сортировки состоит в проверке того, была ли совершена перестановка значений во время любой из итераций внутреннего цикла. Если перестановка не была совершена, то внешний цикл преждевременно завершается (с помощью выражения `break`). Таким образом, если список был изначально отсортирован или для его сортировки требуется лишь несколько итераций, внешнему циклу нет необходимости совершать все запланированные повторения.

```
//  Программа-пример №1 из главы 23
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter23ex1.c
/* Эта программа генерирует 10 случайных чисел,
а затем сортирует их */
#include <stdio.h>
```

Перед сортировкой: 50
32
93
2
74

Во время первого прохода компьютер сравнивает первое и второе значение. Так как 32 меньше 50, они меняются местами:
32
50
93
2
74

После этого компьютер сравнивает 32 и 90 и оставляет их на своих местах. Далее компьютер сравнивает 32 и 2. Так как 2 меньше, 32 и 2 меняются местами:
2
50
93
32
74

Наконец, компьютер сравнивает 2, новое первое значение, и 74 и оставляет их на своих местах.

После первого прохода: 2
50
93
32
74

Во время второго прохода компьютер сравнивает второе значение, 50, и 93 и оставляет их. Далее компьютер сравнивает 50 и 32 и меняет их местами:
2
32
93
50
74

Далее компьютер сравнивает второе значение, 32, и 74 и оставляет их.

После второго прохода: 2
32
93
50
74

Процесс продолжается, пока все числа не будут отсортированы

После третьего прохода: 2
32
50
93
74

После четвертого прохода: 2
32
50
74
93 (отсортировано)

Рис. 23.1. При каждом проходе нижние значения «всплывают» вверх по списку

```
#include <stdlib.h>
#include <time.h>
main()
{
int ctr, inner, outer, didSwap, temp;
int nums[10];
time_t t;
//Если вы не включите это выражение, то программа всегда
//будет генерировать одни и те же 10 чисел
srand(time(&t));
//Первый шаг - заполнить массив случайными числами
//(от 1 до 100)
for (ctr = 0; ctr < 10; ctr++)
{
nums[ctr] = (rand() % 99) + 1);
}
//Распечатать массив в состоянии до сортировки
puts("\nСписок чисел перед сортировкой:");
for (ctr = 0; ctr < 10; ctr++)
{
printf("%d\n", nums[ctr]);
}
//Сортировка массива
for (outer = 0; outer < 9; outer++)
{
didSwap = 0; //Становится равной 1 (ИСТИНА), если список
//еще не сортирован
for (inner = outer; inner < 10; inner++)
{
if (nums[inner] < nums[outer])
{
temp = nums[inner];
nums[inner] = nums[outer];
nums[outer ] = temp;
```

```
didSwap = 1;
}
}
if (didSwap == 0)
{
break;
}
}
//Распечатать массив по состоянию после сортировки
puts("\nСписок чисел после сортировки:");
for (ctr = 0; ctr <10; ctr++)
{
printf("%d\n", nums[ctr]);
}

return 0;
}
```

Экранный вывод программы выглядит следующим образом:

```
Список чисел перед сортировкой:
64
17
1
34
9
5
58
5
6
70
Список чисел после сортировки:
1
5
5
```

```
6
9
17
34
58
64
70
```

ПРИМЕЧАНИЕ

Текст, выведенный программой на экран вашего компьютера может отличаться от приведенного выше, так как при использовании разных компиляторов результаты работы функции `rand()` различаются. Самое важное, на что следует обратить внимание — это список из 10 сгенерированных вашей программой чисел, которые должны быть упорядочены по завершении работы программы.

Механизм перестановки чисел во внутреннем цикле выглядит следующим образом:

```
temp = nums[inner];
nums[inner] = nums[outer];
nums[outer] = temp;
```

Иными словами, если необходима перестановка чисел (первое из сравниваемых двух чисел больше второго), то программа должна поменять местами элементы `nums[inner]` и `nums[outer]`.

Вы можете задаться вопросом, для чего при перестановке значений двух переменных нам потребовалась еще одна, дополнительная, переменная `temp`. Естественный (но неверный) механизм перестановки значений двух переменных выглядел бы, например, следующим образом:

```
nums[inner] = nums[outer]; /* НЕ меняет местами */
nums[outer] = nums[inner]; /* два значения */
```

Первая операция присваивания стирает значение переменной `nums[inner]`, таким образом, второй операции присваивания уже нечего присваивать. Именно поэтому для перестановки значений двух переменных требуется третья переменная.

СОВЕТ

Если вам необходимо отсортировать список по убыванию, то вам нужно лишь поменять знак меньше (<) на знак больше (>) перед кодом, переставляющим значения переменных.

Если вам необходимо расположить в алфавитном порядке список символов, вы можете сделать это, проверяя и переставляя местами значения символьного массива, аналогично тому, как мы сортировали числовые значения в предыдущем примере. В главе 25 вы узнаете, как сохранить перечни символьных данных с возможностью последующей сортировки.

Ускоренный поиск

Иногда сортировка данных может ускорить их поиск. В предыдущей главе вы ознакомились с программой, искавшей идентификационный номер клиента в массиве таких номеров.

Если искомый номер был найден, то для проверки платежеспособности использовался соответствующий этому номеру кредитный баланс (из другого массива). При этом идентификационные номера клиентов не были отсортированы.

Всегда была вероятность, что введенный пользователем идентификационный номер клиента мог не быть найден, например если номер был введен некорректно или номер не был записан в массив. Прежде чем программист понимал, что поиск номера окончился неудачей, программа должна была проверить и сопоставить с введенным каждый из идентификационных номеров, хранящихся в массиве.

Однако, если бы перед началом поиска массивы были отсортированы по идентификационным номерам, то программе не требовалось бы каждый раз проверять каждый элемент массива перед вынесением решения, что искомый элемент не может быть найден. Если бы массив идентификационных номеров был отсортирован, и программе был бы передан номер клиента для поиска, программа практически сразу поняла бы, содержится ли данный элемент в массиве. Рассмотрим следующий список несортированных идентификационных номеров пользователей:

```
313
532
178
```

```
902
422
562
```

Предположим, что программе нужно найти клиента с номером 413. В случае с несортированным массивом, программе пришлось бы сопоставить идентификационный номер 413 с каждым элементом данного массива.

Если бы массив содержал сотни или тысячи элементов, а не только лишь шесть, то компьютеру на осознание того, что совпадений в массиве нет, потребовалось бы гораздо больше времени, так как каждый раз приходилось бы сличать введенный искомый номер со всеми номерами, имеющимися в массиве. Однако если бы значения были отсортированы, то перед пониманием того, что искомый номер не может быть найден, программе не всегда приходилось бы просматривать все элементы массива. Ниже представлен тот же самый список значений, но отсортированный по возрастанию идентификационных номеров:

```
178
313
422
532
562
902
```

Сортировка списка значений позволяет ускорить поиск. Теперь если вы будете искать идентификационный номер 413, ваша программа может завершить поиск, просмотрев только три элемента массива. `422` — третий элемент массива, соответственно, ваша программа может прекратить поиск, так как 422 больше 413, а значит, 422 следует в списке после 413.

ПРИМЕЧАНИЕ

В некоторых случаях поиск по сортированным массивам необязательно быстрее поиска по несортированным. Например, если бы в предыдущем списке вам нужно было бы найти идентификационный номер клиента 998, то ваше программе пришлось бы проверить все шесть элементов массива перед тем, как был бы сделан вывод, что элемент `998` в массиве не найден.

Следующая программа — сочетание программы поиска идентификационных номеров клиента из предыдущей главы и процедуры сортировки значений из предыдущего примера этой главы. Программа производит сортировку идентификационных номеров клиентов, затем запрашивает пользователя ввести номер для поиска. После этого программа определяет, не превышает ли кредитный баланс клиента 100 долларов. Однако, если идентификационный номер в списке не найден, программа завершается раньше. Имейте в виду, что массив только лишь из 10 элементов может создать ощущение, что такая сложная программа — это ненужное излишество, однако код программы, обрабатывающей данные десяток тысяч клиентов был бы точно таким же.

```
//  Программа-пример №1 из главы 23
//  Руководства по С для новичков, 3-е издание
//  Файл Chapter23ex1.c
/* Эта программа осуществляет поиск в отсортированном
списке идентификационных номеров клиентов и выводит
соответствующий кредитный баланс */
#include <stdio.h>
main()
{
int ctr; //Счетчик цикла
int idSearch; //Искомый клиент (ключ)
int found = 0; //Будет равен 1 (ИСТИНА) если клиент
найден
//Определение 10 элементов двух параллельных массивов
int custID[10] = {313, 453, 502, 101, 892,
475, 792, 912, 343, 633};
int custBal[10] = {0.00, 45.43, 71.23, 301.56, 9.08,
192.41, 389.00, 229.67, 18.31, 59.54};
int tempID, inner, outer, didSwap, i; //Для сортировки
float tempBal;
/* Сначала отсортировать массив по номерам */
for (outer = 0; outer < 9; outer++)
{
didSwap = 0; //Становится равной 1 (ИСТИНА), если список
//еще не сортирован
```

```
for (inner = outer; inner < 10; inner++)
{
if (custID[inner] < custID[outer])
{
tempID = custID[inner];
tempBal = custBal[inner];
custID[inner] = custID[outer];
custBal[outer ] = tempBal;
didSwap = 1;
}
}
if (didSwap == 0)
{
break;
}
}

/*Взаимодействие с пользователем, ищущим баланс. */
printf("\n\n*** Проверка баланса клиента ***\n");
printf("Введите искомый номер клиента: ");
scanf(" %d", &idSearch);
/*Поиск и установление наличия номера клиента
в массиве*/
for (ctr=0; ctr<10; ctr++);
{
if (idSearch == custID[ctr]) //Они равны?
{
found = 1; //Да, флаг равенства установить в 1
break; //Продолжать цикл не нужно
}
if (custID[ctr] > idSearch)//Нет смысла продолжать поиск
{
break;
}
```

```
}
//По завершении цикла, номер был либо найден
//(found = 1), либо нет
if (found)
{
if (custBal[ctr] > 100.00)
{
printf("\n**Баланс клиента: $%.2f.\n", custBal[ctr]);
printf("  Кредит недоступен.\n");
}
else
{
printf("\n**У клиента хороший кредитный баланс!\n");
}
}
else
{
printf("**Вы ввели неверный ID клиента.");
printf("\n  Введенный ID %3d в списке не найден.\n",
idSearch);
}
return(0);
}
```

ПРИМЕЧАНИЕ

Кроме покерной программы-игры в приложении В «Программа Покер с обменом», программа из предыдущего примера, пожалуй, самая сложная для понимания из всех программ этой книги. Полное понимание и овладение этой программой переносит вас на новый уровень, значительно выше *новичка*. Наши поздравления и шляпы вверх, когда вы овладеете логикой кода этой программы. Вот видите, в конце концов, программирование на языке С не такое уж и сложное!

Перед ознакомлением с этой программой вы уже овладели методами поиска и сортировки данных в массивах. Эта программа лишь объединяет обе процедуры вместе. Однако единственной дополнительной

задачей, решаемой данной программой, является синхронизация двух параллельных массивов при поиске значений. Как можно видеть в теле блока кода, отвечающего за поиск значений, при перестановке значений элементов массива с идентификационными номерами клиентов (массив `custID`), программа производит поиск по индексу соответствующего элемента массива с данными о балансе клиентов.

Из-за следующего участка кода может произойти преждевременное завершение процедуры поиска:

```
if (custID[ctr] > idSearch)//Нет смысла продолжать
поиск
{
break;
}
```

В случаях, когда массив содержит несколько тысяч элементов, такой оператор `if` может сэкономить значительное количество процессорного времени.

Содержать сортированные массивы не всегда просто и эффективно. Например, вряд ли вы захотели бы, чтобы ваша программа сортировала большой массив каждый раз при добавлении, изменении или удалении определенного значения.

После сортировки нескольких тысяч элементов массива повторная сортировка массива после добавления новых значений может занять продолжительное время, даже если у вас быстрый компьютер. Усовершенствованные методы работы с массивами позволяют гарантировать, что вы вводите новые элементы массива уже в порядке следования, и массив остается отсортированным. (Однако такие техники находятся далеко вне поля зрения данной книги.) Вы и так хорошо прогрессируете, поэтому усложнять этот процесс мы не станем.

Абсолютный минимум

Целью данной главы было ознакомить вас с методом пузырьковой сортировки и упорядочения элементов массива. Для сортировки массивов нет необходимости в изучении новых команд языка С. Возможность сортировки — одно из основных преимуществ массивов. При необходимости нумерация элементов массива позволяет вам проходить по каж-

дому из них и переставлять значения. Ниже перечислены основные концепции текста главы.

- При сортировке значений от меньшего к большему воспользуйтесь методом сортировки по возрастанию.
- При сортировке значений от большего к меньшему воспользуйтесь методом сортировки по убыванию.
- Вложенный цикл `for` идеально подходит для осуществления пузырьковой сортировки, как вы могли убедиться благодаря примерам в этой главе.
- Не меняйте местами значения двух переменных без предоставления третьей переменной для хранения промежуточного значения.
- Процедуры сортировки элементов массива не обязательно сложны: начните с применения простых методик, описанных в этой главе, и адаптируйте их под свои нужды.
- Не забывайте сортировать массивы: это ускорит процесс поиска содержащихся в них значений.

Глава 24
РАЗГАДКА ТАЙНЫ УКАЗАТЕЛЕЙ

В этой главе

- Работа с адресами памяти
- Объявление переменных-указателей
- Использование разыменовывающей *

Переменные-указатели, часто называемые просто *указателями*, языка C позволяют вам сделать гораздо большее по сравнению с возможностями других языков программирования, не поддерживающих указатели. При первом знакомстве с указателями вы, возможно, зададитесь вопросом: «А в чем смысл?» (даже после того, как вы в полной степени овладеете указателями, вы, вероятно, будете продолжать задавать тот же самый вопрос). Указатели предоставляют средства для поистине мощного программирования на языке C. В этой книге мы покажем вам лишь самую верхушку айсберга указателей, те концепции, которые вы изучите здесь, станут прочным фундаментом вашего будущего в программировании на C.

Адреса памяти

В вашем компьютере есть большое количество памяти. Во время выполнения ваша программа находится в памяти компьютера, в памяти также содержатся и переменные вашей программы. У каждого дома есть свой адрес, свой адрес есть и у каждой ячейки памяти. Это не совпадение, что у каждой ячейки памяти есть собственный *адрес*. Как и в случае с адресами домов, адреса участков памяти уникальны: в памяти не существует двух ячеек с одинаковыми адресами. Память компьютера подобна аппаратному массиву, в котором ячейка памяти — это элемент, а адрес — индекс этого элемента в массиве.

Когда вы объявляете переменные, компилятор находит свободное место в памяти и присваивает этому свободному участку памяти ука-

занное вами имя. И это хорошо. Вместо того, чтобы запоминать, что номер заказа хранится в ячейке памяти с адресом 34532, вам нужно лишь запомнить имя `orderNum` (при условии, что при объявлении вы присвоили переменной имя `orderNum`). Имя переменной `orderNum` *гораздо* проще запомнить, чем номер.

Объявление переменных-указателей

Как и в случае с переменными других типов, перед использованием указателей вы должны их объявить. Прежде чем мы пойдем дальше, вы должны познакомиться с двумя новыми операторами.

В табл. 24.1 мы привели эти операторы вместе с описанием.

Табл. 24.1. Операторы для работы с указателями

Оператор	Описание
&	Операция взятия адреса
*	Операция разыменования

Вы уже встречали оператор `*` раньше. Откуда компилятор узнает, в каком случае данный оператор выполняет операцию умножения, а в каком — разыменования? Контекст использования оператора определяет то, каким образом данный оператор будет интерпретирован компилятором.

Вы также видели оператор `&` перед переменными в функции `scanf()`. В функции `scanf()` оператор `&` выполняет операцию взятия адреса. Функция `scanf()` требует передачи адреса немассивных переменных.

Далее приведен пример объявления целочисленной переменной и переменной типа `float`:

```
int num;
float value;
```

Для объявления целочисленного указателя или указателя типа `float`, просто вставьте оператор `*`:

```
int * pNum; /*Объявление двух указателей*/
float * pValue;
```

ПРИМЕЧАНИЕ

Для имен указателей не существует никаких специальных ограничений, однако многие программисты, работающие на языке C, предпочитают использовать приставку `p` с именами указателей (от английского слова «pointer» — указатель. — *Примеч. пер.*), как сделали и мы, но вы можете называть указатели любыми именами. Приставка `p` просто служит вам напоминанием, что перед вами указатель, а не обычная переменная.

Каждому типу данных соответствует свой тип указателя. В языке C предусмотрены символьные указатели, указатели длинных целых чисел и так далее. Переменные-указатели содержат адреса других переменных. В этом их основная цель. Для присвоения указателю адреса переменной необходимо воспользоваться оператором `&`. До того как вы присвоите указателю адрес какой-либо переменной, указатель считается неинициализированным и не может быть использован в программе.

В следующем участке кода мы объявляем целочисленную переменную `age` и записываем в нее значений `19`. Далее объявляется указатель `pAge` и инициализируется таким образом, чтобы указывать на переменную `age`. Оператор взятия адреса читается так, как подсказывает его название. Вторая строка кода говорит компилятору записать адрес переменной `age` в указатель `pAge`:

```
int age = 19; /*Запись числа 19 в переменную age*/
int * pAge = &age; /*Связь с указателем*/
```

Вы не можете знать, в ячейке, с каким адресом будет сохранена переменная `age`. Однако, какой бы адрес ни был назначен компилятором, он будет записан в переменную-указатель `pAge`. Когда переменная указатель содержит адрес другой переменной, говорят, что она *указывает* на эту переменную. Предположив, что переменная `age` сохранена в ячейке с адресом 18826 (только компилятор знает, где конкретно эта переменная была сохранена), рис. 24.1 показывает, как схематически выглядит память компьютера.

ВНИМАНИЕ

Только тот факт, что вы объявили две переменные друг за другом, совсем не означает, что они будут сохранены в соседних ячейках памяти. Компьютер *может* сохранить их вместе, а может и не сохранить.

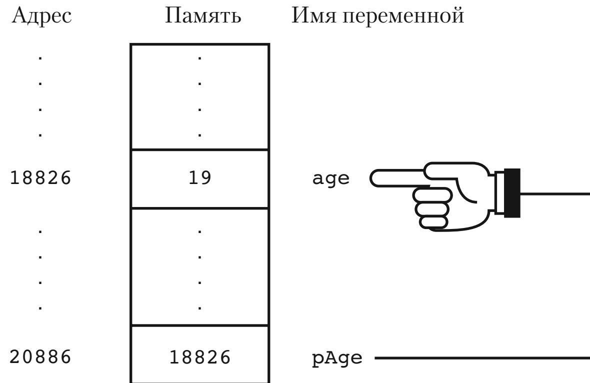


Рис. 24.1. Переменная `pAge` указывает на переменную `age`, если `pAge` содержит адрес `age`»

ВНИМАНИЕ

Никогда не пытайтесь задать адрес переменной одного типа указателю другого типа. По правилам языка C адрес переменной любого типа может быть присвоен только указателю этого же типа.

Звездочка (*) не является частью имени переменной-указателя. *Оператор разыменования* * может быть использован для разных целей, но при объявлении указателя данный оператор нужен только лишь для того, чтобы сообщить компьютеру, что данная переменная является указателем. Следующие четыре строки кода *абсолютно аналогичны* двум выражениям из предыдущего примера. Обратите внимание, что при сохранении значения-адреса переменной в указателе оператор * не используется, кроме случаев, когда вы одновременно с сохранением значения также объявляете указатель:

```
int age; //Объявление обычной переменной
int * pAge; //Объявление указателя на целочисленную
переменную
int age = 19; //Запись числа 19 в переменную age
pAge = &age; //Связь с указателем
```

Использование оператора разыменования *

Как только вы привязали указатель к переменной, вы можете работать со значением этой переменной, *разыменовав* указатель. Програм-

мисты никогда не используют простых слов, когда можно использовать сложное, да вдобавок еще и способное запутать большинство простых обывателей. Разыменование означает, что вы используете указатель для получения доступа к значению другой переменной. Для разыменования используется оператор разыменования `*`.

Иными словами, существует два способа изменения значения переменной `age` (разумеется, если все переменные объявлены именно так, как описано выше):

```
age = 25;
```

и

```
*pAge = 25; /*Сохраняет 25 там, где указано pAge */
```

Такая операция присваивания сообщает компилятору сохранить значение 25 по адресу, указываемому переменной `pAge`. Аналогичным образом значение переменной может быть использовано:

```
printf("Возраст: %d.\n", age);
```

и

```
printf("Возраст: %d.\n", *pAge);
```

Когда функция работает с передаваемым в качестве аргумента указателем, используется оператор разыменования. В главе 32 вы узнаете о том, как передавать указатели функциям. Если функция использует переменную-указатель, посылаемую ей другой функцией, то оператор разыменования должен использоваться по всему коду перед каждым вхождением имени данного указателя.

Оценить истинную мощь указателей вы сможете при чтении глав, посвященных функциям, однако ознакомление с указателями уже сейчас также имеет смысл. Далее приведена простая программа, в которой объявляются целочисленная, символьная переменная, переменная типа float, а также их аналоги-указатели:

```
//  Программа-пример №1 из главы 24
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter24ex1.c
/* Эта программа демонстрирует указатели с помощью
объявления и инициализации обычных переменных
```

```
и переменных-указателей типа int, float и char,
а затем отображает значения каждой из переменных*/
#include <stdio.h>
main()
{
int kids;
int * pKids;
float price;
float * pPrice;
char code;
char * pCode;
price = 17.50;
pPrice = &price;
printf("\nСкольких детей вы ведете в аквапарк? ");
scanf(" %d", &kids);
    pKids = &kids;
    printf("\nУ Вас есть скидка?");
    printf("\nВведите E для скидки сотрудника, S для
скидки по программе Sav-More ");
    printf("или N если скидки нет: ");
    scanf(" %c", &code);
    pCode = &code;
    printf("\nСначала давайте поработаем
с переменными:\n");
    printf("У вас %d детей...\n", kids);
    switch (code) {
    case ('E') :
        printf("Скидка сотрудника позволяет сэкономить
25%% от цены ");
        printf("$%.2f price", price);
        printf("\nОбщая стоимость билетов: $%.2f",
(price*.75*kids));
        break;
    case ('S') :
```

```
        printf("Скидка Sav-more позволяет сэкономить
15%% от цены ");
        printf("$%.2f price", price);
        printf("\n Общая стоимость билетов: $%.2f",
(price*.85*kids));
        break;
    default : // Если введена N при отсутствии скидки
              // или неправильная буква
        printf("Вы заплатите полную стоимость билетов,
");
        printf("которая составит $%.2f", price);
    }
    // Теперь повторим тот же самый код, но
использовав
    // для получения тех же результатов разыменованные
указатели
    printf("\n\n\nТеперь поработаем
с указателями:\n");
    printf("У вас %d детей...\n", *pKids);
    switch (*pCode) {
    case ('E') :
        printf("Скидка сотрудника позволяет сэкономить
25%% от цены ");
        printf("$%.2f price", *pPrice);
        printf("\nОбщая стоимость билетов: $%.2f",
(*pPrice *.75 * *pKids));
        break;
    case ('S') :
        printf("Скидка Sav-more позволяет сэкономить
15%% от цены ");
        printf("$%.2f price", *pPrice);
        printf("\nОбщая стоимость билетов: $%.2f",
(*pPrice *.85 * *pKids));
        break;
    default : // Если введена N при отсутствии скидки
```

```
            // или неправильная буква
    printf("Вы заплатите полную стоимость билетов, ");
    printf("которая составит $%.2f", *pPrice);
    }
    return(0);
}
```

Далее приведен результат тестового запуска программы:

```
Скольких детей вы ведете в аквапарк? 3
У Вас есть скидка?
Введите E для скидки сотрудника, S для скидки по
программе Sav-More или N если скидки нет: S
Сначала давайте поработаем с переменными:
У вас 3 детей...
Скидка Sav-more позволяет сэкономить 15%% от цены
$17.50
Общая стоимость билетов: $44.63
Теперь поработаем с указателями:
У вас 3 детей...
Скидка Sav-more позволяет сэкономить 15%% от цены
$17.50
Общая стоимость билетов: $44.63
```

В этой программе нет ничего революционного или запредельно сложного. Цель этого примера в том, чтобы вы ознакомились и привыкли к указателям, в том числе к объявлению, настройке и ссылке на указатели любого вида. Повторимся, что указатели вам будут требоваться постоянно, когда вы начнете использовать функции для взятия и возвращения данных.

Абсолютный минимум

Целью этой главы было дать вам начальные представления о переменных-указателях. Переменная-указатель — это переменная, значением которой является местоположение другой переменной. Вы можете ссылаться на переменную либо по имени, либо с помощью разыменованного указателя.

В языке С указатели используются довольно часто, особенно в программировании более сложных систем. Как вы узнаете в следующей главе, указатели — ничто иное, как массивы в маске. Так как указатели предоставляют больше гибкости, чем массивы, многие программисты, работающие на языке С, прекращают использование массивов, в полной мере овладев указателями. Далее приведены основные концепции данной главы.

- Привыкайте к адресам ячеек памяти, так как адреса — это основа использования указателей.
- Для получения адреса переменной воспользуйтесь оператором `&`.
- Для объявления переменной-указателя и ее разыменования воспользуйтесь оператором `*`. `*pAge` и `age` ссылаются на одну и ту же ячейку памяти, в том случае, если `pAge` ссылается на `age`.
- Не пытайтесь заставить указатель одного типа ссылаться на переменную другого типа.
- Не беспокойтесь о *точном* адресе ячейки, в которой была сохранена переменная. Если вы воспользуетесь оператором `&`, компьютер позаботится обо всем остальном.
- Не забывайте использовать оператор `*` при разыменовании указателя, иначе вы получите неверное значение.
- Не забегайте слишком вперед. Вы в полной мере сможете оценить указатели только получив определенный опыт программирования на языке С. На данном этапе указатели вполне могут показаться вам абсолютно бесполезными. Единственное, что может вас порадовать — это точное знание того, для чего в функции `scanf()` необходимо использовать оператор `&`.

Глава 25
МАССИВЫ И УКАЗАТЕЛИ

В этой главе

- Осознание того, что имена массивов — это указатели
- Переход вниз по списку
- Работа с символами и указателями
- Соблюдаем осторожность с длиной строк
- Создание массивов указателей

В этой главе мы расскажем вам о том, что в языке C массивы и переменные-указатели объединяет много общих признаков. Вообще, на самом деле массив — это особый вид указателя. Благодаря сходствам вы можете воспользоваться нотацией указателей для получения значений элементов массива и нотацией массива для получения значения, на которое ссылается указатель.

Возможно, самой важной причиной, для чего необходимо узнать о том, как взаимодействуют и перекликаются массивы и указатели, является необходимость обработки строк символов. Сочетая нотацию указателей (используя оператор разыменования) и нотацию массивов (используя индексы), вы можете сохранять списки символьных строк и ссылаться на них так же просто, как вы ссылались бы на элементы массива любого другого типа данных.

После того, как вы изучите *динамическую память* (heap) — специальный участок памяти, который мы представим вам в следующей главе, вы поймете, что указатели — это единственный способ получить доступ к динамической памяти, куда были записаны ваши данные.

Названия массивов и указатели

Имя массива — ничто иное, как указатель на первый элемент этого массива. Впрочем, нельзя сказать, что имя массива — это *переменная-*

указатель. Имена массивов принято называть *константами-указателями*. Следующее выражение объявляет и инициализирует целочисленный массив:

```
int vals[5] = {10, 20, 30, 40, 50};
```

Вы можете сослаться на массив по индексной нотации. Это вы уже и так знаете. Однако язык C не только прикрепляет индексы к значениям в памяти, но также настраивает указатель на массив и называет его `vals`.

Вы не можете изменить значение указателя `vals`: он подобен фиксированной переменной-указателю, значение которой заблокировано компилятором. На рис. 25.1 показано, что на самом деле происходит, когда вы объявляете и инициализируете массив `vals`.

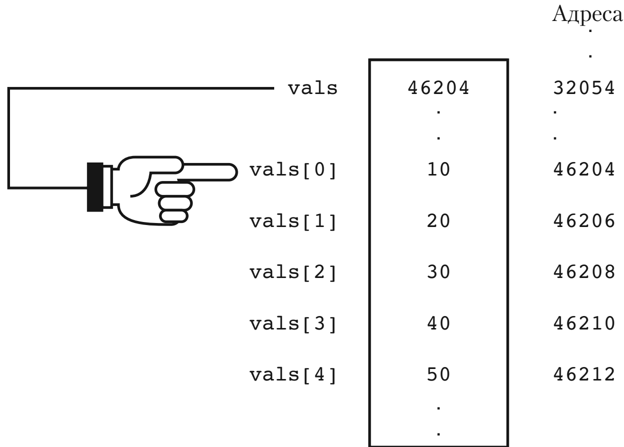


Рис. 25.1. Имя массива — это указатель на первый элемент этого массива

Так как имя массива — это указатель (значение которого изменить нельзя), вы можете распечатать значение первого элемента массива следующим образом:

```
printf("Значение первого элемента массива: %d.\n",
vals[0]);
```

Однако более важным в данной главе является то, что вы можете вывести на печать первый элемент массива также следующим способом:

```
printf("Значение первого элемента массива: %d.\n",
*vals);
```

Как вы увидите совсем скоро, следующее выражение также эквивалентно двум предыдущим и дает доступ к элементу массива `vals[0]`:

```
printf("Значение первого элемента массива: %d.\n",
*(vals+0));
```

ПРИМЕЧАНИЕ

Именно из-за того, что массив — это фиксированная константа-указатель, вы не можете просто так поместить имя массива слева от знака =. Константу изменить нельзя. (Впрочем, запомните, что язык С делает послабление к этому правилу в случае, когда вы только объявили массив, так как компьютеру еще лишь предстоит зафиксировать массив по определенному адресу.)

Переход вниз по списку

Так как имя массива — это указатель на *первый* элемент данного массива, если вы хотите получить доступ ко второму элементу, вам нужно лишь прибавить `1` к имени массива и разыменовать *данный* участок памяти. Данный набор вызовов функций `printf()`:

```
printf("Значение первого элемента массива: %d.\n",
vals[0]);
printf("Значение второго элемента массива: %d.\n",
vals[1]);
printf("Значение третьего элемента массива: %d.\n",
vals[2]);
printf("Значение четвертого элемента массива: %d.\n",
vals[3]);
printf("Значение пятого элемента массива: %d.\n",
vals[4]);
```

абсолютно аналогичен следующему набору:

```
printf("Значение первого элемента массива: %d.\n",
*(vals + 0));
printf("Значение второго элемента массива: %d.\n",
*(vals + 1));
printf("Значение третьего элемента массива: %d.\n",
*(vals + 2));
```

```
printf("Значение четвертого элемента массива: %d.\n",
*(vals + 3));
printf("Значение пятого элемента массива: %d.\n",
*(vals + 4));
```

Если `vals` — это константа-указатель (и это действительно так), и константа-указатель содержит число, являющееся адресом участка памяти, где находится первый элемент массива, то прибавление 1 или 2 (или любого другого значения) к `vals` перед разыменованием добавляет 1 или 2 к адресу, на который указывает `vals`.

> **СОВЕТ**
>
> Если вы не можете понять, в чем смысл всей этой неразберихи, то наберитесь терпения, вскоре вы увидите, что нотация указателей языка C позволяет вам работать с языком C *почти* так, словно бы в C были строковые переменные.

Как вы можете помнить, целые числа, как правило, занимают памяти больше, чем 1 байт. В предыдущих вызовах функции `printf()` к адресам внутри массива `vals` прибавлялась 1 для получения адреса следующего разыменованного участка памяти, но язык C здесь вам помогает.

При прибавлении `1` к целочисленному указателю компьютер автоматически прибавляет к нему `1` размер типа `int` (или один размер типа `double` при прибавлении `1` к указателю типа `double` и так далее).

Выражение `*(vals + 2)` сообщает компьютеру, что вы хотите получить доступ к *третьему целому числу* из списка чисел, на которые ссылается `vals`.

Символы и указатели

Следующие две строки практически одинаково выглядят в памяти компьютера, с той лишь разницей, что во втором выражении `pName` — *переменная*-указатель, а не константа-указатель:

```
char name[] = "Andrew B. Mayfair"; /* name указывает
на A */
char *pName = "Andrew B. Mayfair"; /* pName указывает
на A */
```

Так как `pName` — переменная-указатель, вы *можете* поместить ее слева от знака равенства! Таким образом, вам не всегда необходимо пользоваться функцией `strcpy()`, когда вы хотите присвоить символьному указателю новое значение. Символьный указатель всегда указывает только на первый символ в строке. Однако код преобразования `%s` и все остальные строковые функции точно также могут быть с легкостью использованы, как и с символьными массивами (обе сущности по сути идентичны), так как все эти функции «знают», что они должны остановиться при достижении нуль-символа.

Для того, чтобы записать новое имя в массив `name`, вы должны воспользоваться функцией `strcpy()` для посимвольной записи строки в массив, однако, чтобы заставить указатель `pName` ссылаться на другое имя, вам нужно сделать следующее:

```
pName = "Theodore M. Brooks";
```

СОВЕТ

Единственная причина, по которой присвоение строк работает в языке С, заключается в том, что язык С распределяет строковые литералы вашей программе по участкам памяти, а затем *заменяет* их в программе на адреса тех участков памяти, в которых литералы были сохранены. На самом деле строка `"Theodore M. Brooks"` не записывается в указатель `pName`, так как `pName` может содержать только адреса. Компилятор записывает *адреса* символов строки `"Theodore M. Brooks"` в указатель `pName`.

Теперь вы знаете способ присвоения новых строковых значений без использования функции `strcpy()`. Чтобы узнать этот способ, вам пришлось проделать немалый путь, но разве вы не рады, что наконец узнали его? Если рады, то присядьте: здесь есть нюанс (ну куда же без него?).

Будьте внимательны с длиной

Сохранение строк литералов в символьных массивах только что описанным способом считается нормальной практикой. Новые строки, присваеваемые с помощью знака =, могут быть короче или длиннее предыдущих строк. И это хорошо, так как, если помните, вы не можете сохранить в символьном массиве строку длиннее изначально зарезервированной строки.

Однако вы должны быть предельно внимательны и *не* позволять *программе* сохранять с помощью символьных указателей длинней, чем первая строка, на которую ссылался указатель. Это может быть несколько сложно, но продолжайте чтение: мы упростили и сократили эту главу насколько возможно. После инициализации символьной переменной-указателя следующим образом:

```
main()
{
char * name = "Tom Roberts";
/* Продолжение программного кода */
```

никогда не позволяйте пользователю вводить новую строку с помощью функции `gets()`, например, так:

```
gets(name); /* Не очень безопасно */
```

Проблема с этим выражением в том, что пользователь может ввести строку, длина которой превышает длину строки `Tom Roberts` — первой строки, присвоенной символьному указателю. Несмотря на то, что символьный указатель может ссылаться на строки любой длины, функция `gets()`, а также другие функции, например, `scanf()`, `strcpy()` и `strcat()`, «не знают», что в качестве аргумента им передается символьный указатель. Так как данным функциям может быть передан символьный массив, изменить место хранения которого нельзя, то перечисленные функции напрямую записывают новую строку поверх старой, уже хранящейся в `name`. Если пользователь ввел строку, длина которой превышает длину строки, на которую указывает `name`, то может произойти перезапись других данных.

ВНИМАНИЕ

Да, все это довольно сложно. Возможно, вам придется позднее перечитать этот раздел, когда вы привыкните к массивам и указателям и начнете лучше в них разбираться.

Впрочем, есть небольшой трюк, позволяющий сохранить преимущество символьного указателя и заключающейся в возможности записи строк литералов в указатель и сохранении подушки безопасности, предоставляемой массивами, благодаря которой указатель может быть использован для получения пользовательского ввода.

Если вы хотите сохранить пользовательский ввод в виде строки, на которую ссылается указатель, сначала вы должны зарезервировать достаточное количество места для хранения такой вводимой с клавиатуры строки.

Самый простой способ заключается в резервировании символьного массива и последующем присвоении первому элементу этого массива символьного указателя:

```
char input[81]; //Хранит строку длиной в 80 символов
char *iptr = input; //Аналогично выражению char *iptr
= input[0]
```

Теперь вы можете вводить строку с помощью указателя, но только до тех пор, пока длина введенной пользователем строки не превышает 81 байт:

```
gets(iptr); /* Гарантирует, что *iptr указывает на
введенную пользователем строку */
```

Чтобы быть до конца уверенным в том, что длина вводимых строк не превысит 81 символ включая нуль-символ, вы можете воспользоваться вполне приятной функцией `fgets()`, ограничивающей количество символов, принимаемое программой от пользователя. Функция `fgets()` работает точно также, как функция `gets()`, за исключением того, что вы также указываете в качестве аргумента длину строки. Следующее выражение демонстрирует функцию `fgets()` в действии:

```
fgets(iptr, 81, stdin); /* Получает до 80 символов
и прибавляет нуль-символ */
```

Второе значение — максимальное количество символов, которые вы хотели бы сохранить из пользовательского ввода. Всегда оставляйте одно свободное место для нуль-символа, завершающего строку. Указатель `iptr` может ссылаться на строку длиной максимум в 81 символ. Если пользователь вводит строку, длина которой меньше 81 символа, то указатель `iptr` будет ссылаться на нее безо всяких проблем.

Однако если пользователь начинает сходить с ума и вводит строку длиной в 200 символов, то указатель `iptr` будет ссылаться только на первые 80 символов и нуль-символ, занимающий 81-ю позицию в строке и прибавляемый функцией `fgets()`. Оставшаяся часть введенного текста программой игнорируется.

СОВЕТ

Вы также можете пользоваться функцией `fgets()` для чтения строк из файлов данных. Третьим аргументом функции `fgets()` может быть указатель на файл на диске, но вы познакомитесь с указателями на файл позже в этой книге. На данный момент, указывайте `stdin` в качестве третьего аргумента с тем, чтобы функция обращалась за вводом данных к клавиатуре, а не куда-либо еще.

Вы также можете присвоить указателю строку литералов, воспользовавшись оператором присваивания следующим образом:

```
iptr = "Mary Jane Norman";
```

Массивы указателей

Если вам необходимо воспользоваться большим количеством указателей, создайте массив указателей. Массив указателей также легко объявить, как и массив любого другого типа данных, за исключением того, что вам нужно добавить оператор * после названия типа данных. Следующие выражения резервируют массив из 25 целочисленных указателей и массив из 25 символьных указателей:

```
int * ipara[25]; /* 25 целочисленных указателей */
char * cpara[25]; /* 25 символьных указателей */
```

Массив символов, пожалуй, самый интересный случай, так как вы можете хранить в нем целый список строк. Если точнее, то вы можете *указывать* на разные строки. Следующая программа иллюстрирует две вещи: как инициализировать массив строк в момент объявления и как вывести их на печать с помощью цикла `for`:

ПРИМЕЧАНИЕ

На самом деле программа решает несколько большее количество задач, чем указано выше. Данная программа также позволяет вам выставить рейтинг девяти строкам (в данном случае, это англоязычные названия просмотренных вами кинофильмов) по шкале от 1 до 10, а затем использует знакомую нам пузырьковую сортировку, но вместо упорядочивания по возрастанию программа сортирует список строк по рейтингу от большего к меньшему. Ничего страшного нет в возвращении назад и добавлении ранее узнанных концепций при изучении новых уроков, именно так вы начнете создавать хорошо отлаженные и интересные программы!

```
//  Программа-пример №1 из Главы 25
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter25ex1.c
/* Эта программа объявляет и инициализирует
массив символьных указателей, а затем запрашивает
соответствующие рейтинги */
#include <stdio.h>
main()
{
    int i;
    int ctr = 0;
    char ans;
    //Объявление 9 символьных указателей
и инициализация их
    char * movies[9] = {"Amour", "Argo",
                        "Beasts of the Southern Wild",
                        "Django Unchained",
                        "Les Miserables",
                        "Life of Pi", "Lincoln",
                        "Silver Linings Playbook",
                        "Zero Dark Thirty"};
    int movieratings[9]; // Соответствующий массив 9
целых чисел
                            // для рейтинга кинофильмов
    char * tempmovie = "Будет использовано для
сортировки оцененных фильмов";
    int outer, inner, didSwap, temprating; // для
цикла сортировки
    printf("\n\n*** Номинация на Оскар 2012 года!
***\n\n");
    printf("Пора оценить лучших кандидатов на
премию:");
    for (i=0; i< 9; i++)
    {
        printf("\nВы видели %s? (Y/N): ", movies[i]);
```

```
        scanf(" %c", &ans);
        if ((toupper(ans)) == 'Y')
        {
            printf("\nКаков ваш рейтинг по шкале ");
            printf("от 1 до 10: ");
            scanf(" %d", &movieratings[i]);
            ctr++; // Будет использовано только для
печати
                   // просмотренных вами фильмов
            continue;
        }
        else
        {
            movieratings[i] = -1;
        }
    }
    // Теперь отсортируем фильмы по рейтингу
    // (несмотренные фильмы пойдут вниз)
    for (outer = 0; outer < 8; outer++)
    {
        didSwap = 0;
        for (inner = outer; inner < 9; inner++)
        {
            if (movieratings[inner] >
movieratings[outer])
            {
                tempmovie = movies[inner];
                temprating = movieratings[inner];
                movies[inner] = movies[outer];
                movieratings[inner] =
movieratings[outer];
                movies[outer] = tempmovie;
                movieratings[outer] = temprating;
                didSwap = 1;
```

```
            }
        }
        if (didSwap == 0)
        {
            break;
        }
    }
    // Теперь вывести просмотренные фильмы по порядку
    printf("\n\n** Ваши рейтинги фильмов-кандидатов
");
    printf("на премию Оскар 2012 **\n");
    for (i=0; i < ctr; i++)
    {
        printf("%s оценен на %d!\n", movies[i],
movieratings[i]);
    }
    return(0);
}
```

Далее приведен результат тестового запуска программы:

```
*** Номинация на Оскар 2012 года!
Пора оценить лучших кандидатов на премию:
Вы видели Amour? (Y/N): Y
Каков ваш рейтинг по шкале от 1 до 10: 6
Вы видели Argo? (Y/N): Y
Каков ваш рейтинг по шкале от 1 до 10: 8
Вы видели Southern Wild? (Y/N): N
Вы видели Django Unchained? (Y/N): Y
Каков ваш рейтинг по шкале от 1 до 10: 7
Вы видели Les Miserables? (Y/N): Y
Каков ваш рейтинг по шкале от 1 до 10: 7
Вы видели Life of Pi? (Y/N): N
Вы видели Lincoln (Y/N): Y
Каков ваш рейтинг по шкале от 1 до 10: 6
```

```
Вы видели Silver Linings Playbook? (Y/N): Y
Каков ваш рейтинг по шкале от 1 до 10: 9
Вы видели Zero Dark Thirsty? (Y/N): N
** Ваши рейтинги фильмов-кандидатов на премию Оскар
2012 **
Argo оценен на 8.
Les Miserables оценен на 7.
Django Unchained оценен на 7.
Lincoln оценен на 6.
Amour оценен на 6.
```

На рис. 25.2 изображено, каким образом программа организует массив movies в памяти компьютера. Каждый элемент — не что иное, как символьный указатель, содержащий адрес названия фильма. Важно понимать, что массив movies не содержит строк: только указатели на строки.

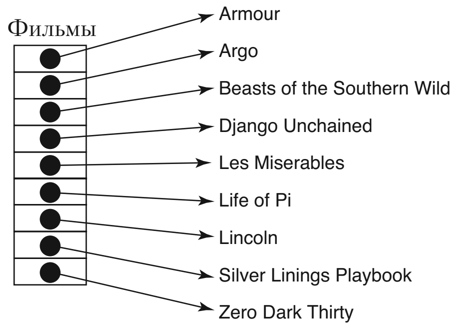


Рис. 25.2. Массив movies содержит указатели на строки

Видите, хотя в языке C строковых массивов и не существует (так как строковые переменные в языке C не предусмотрены), сохранение строковых указателей в массиве movies заставляет программу вести себя так, словно массив movies был строковым.

После этого программа с помощью цикла проходит по названиям девяти фильмов в массиве и спрашивает пользователя, видел ли он каждый из этих фильмов. Если ответ пользователя Y (или y после конвертации с помощью функции toupper()), программа запрашивает целое число в диапазоне от 1 до 10. Кроме того, происходит увеличение

счетчика (ctr), таким образом программа точно знает, сколько фильмов пользователь посмотрел. Если ответ N (или любой другой символ, отличающийся от символов Y и y), то данному фильму присваивается рейтинг –1, таким образом, данный фильм окажется в конце списка при сортировке.

После того как программа завершит сопоставление фильмов и рейтингов, для сортировки фильмов от лучшего к худшего по мнению пользователя используется пузырьковая сортировка. Ведь приятно знать, что вы можете использовать процедуру сортировки применительно к строковым массивам? Теперь отсортированный массив готов к выводу на печать. Однако цикл for будет повторен только ctr количество раз, что позволит не выводит на экран названия фильмов, которые пользователь не видел.

Абсолютный минимум

Целью данной главы было спровоцировать вас на размышления о том, чем и насколько похожи массивы и указатели. На самом деле имя массива — это лишь указатель на первый элемент данного массива. В отличие от переменных-указателей, значение имени массива изменить нельзя. В этом заключается основная причина того, что имя массива не может появляться слева от знака равно.

Использование указателей предоставляет больше гибкости, чем использование массивов. Вы можете напрямую присвоить строку литералов переменной-указателю, в то время как для присвоения строки массиву вы должны воспользоваться функцией strcpy(). В своей карьере программиста на языке С вы найдете немало применений переменным-указателям. Далее перечислены основные концепции данной главы:

- Для присваивания строк литералов напрямую используйте символьные указатели.
- Для получения доступа к значениям указателей или элементов массива пользуйтесь либо нотацией массивов, либо операцией разыменования, свойственной указателям.
- Не используйте встроенную функцию для заполнения участка памяти, на который ссылается символьный указатель, если только данный указатель изначально не был настроен на хранение длинных строк.

Глава 26

МАКСИМИЗАЦИЯ ПАМЯТИ ВАШЕГО КОМПЬЮТЕРА

В этой главе

- Размышления о динамической памяти
- Понимание, для чего вам необходима динамическая память
- Выделение динамической памяти
- Принятие мер при нехватке динамической памяти
- Освобождение динамической памяти
- Управление множественными операциями выделения памяти

Не только абсолютные новички в программировании на языке C могут найти содержание данной главы несколько непонятным. Даже опытные программисты зачастую теряются при работе с динамической (динамически распределяемой*) памятью. Этим термином скрывается неиспользованная память вашего компьютера. Свободная память (после занятия памяти вашей программой, переменными вашей программы и рабочим пространством операционной системы) составляет доступную динамическую память вашего компьютера, как показано на рис. 26.1.

Зачастую вам будет требоваться доступ к динамической памяти потому, что вашей программе будет требоваться памяти больше, чем вы изначально объявили с помощью переменных и массивов. В этой главе мы расскажем вам, почему и как нужно использовать динамически распределяемую память вместо переменных.

Вы не можете присвоить имена переменных динамической памяти. Единственный способ получения доступа к данному виду памяти — это указатели. Разве вы не рады, что уже изучили указатели? Не изучив указатели, вы не смогли бы заняться изучением динамической памяти.

* Дословно термин «heap» с английского языка переводится как «куча». — *Примеч. пер.*

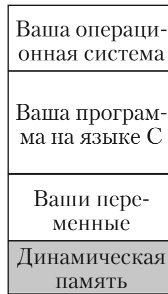


Рис. 26.1. Динамическая память — это неиспользованная память

ПРИМЕЧАНИЕ

Свободную динамическую память также принято называть нераспределенной динамической памятью. Часть динамической памяти, используемая вашей программой в любой момент времени, называется распределенной динамической памятью. Во время выполнения программы количество используемой динамической памяти может варьироваться. До настоящего момента ни одна из программ, приведенных в данной книге, не использовала динамическую память.

Размышления о динамической памяти

Теперь, когда вы узнали, что такое динамическая память — неиспользованный раздел контигуальной памяти, — забудьте о том, чему мы вас научили! Вы гораздо быстрее научитесь пользоваться динамической памятью, если будете воспринимать ее как одну большую кучу памяти, сложенную в большой сугроб. Объяснение — в следующем абзаце.

Во время выполнения программы вы будете то распределять (использовать), то высвобождать динамическую память. Когда вы запрашиваете динамическую память, вы не можете знать, откуда именно из кучи придет нужная вам память. Таким образом, если одно выражение кода вашей программы забирает часть динамической памяти, а следующее за ним выражение также забирает часть динамической памяти, это совсем не значит, что физически второй раздел выделенной памяти будет находиться по соседству с первым.

Вышеописанное аналогично черпанию совковой лопаты грязи из большой кучи. Вы не зачерпываете на совок куски грязи, которые находились точно под кусками грязи, зачерпнутыми в предыдущий раз. Кроме того, если вы бросите совок грязи обратно в кучу, куски грязи не

вернутся ровно туда же, откуда были зачерпнуты. Хотя подобная аналогия кажется преувеличивающей концепции компьютерной памяти, вскоре вы обнаружите, что будете лучше понимать механизмы работы с динамической памятью, если будете думать об этой памяти как о большой куче грязи. Аналогия состоит в том, что вы не знаете, откуда именно берется запрашиваемая память и куда именно она будет возвращена при высвобождении. Вы знаете только, что динамическая память приходит из кучи и уходит обратно.

Если вы запросите распределение 10 байтов динамической памяти, то эти 10 байтов обязательно будут смежными. Однако важно знать и то, что следующий участок выделяемой динамической памяти не обязательно будет следовать сразу же за предыдущим, поэтому на практике вам не стоит на это полагаться.

Динамическая память используется не только вашей программой, но также и операционной системой компьютера. Если вы работаете на компьютере, подключенном к сети, или используете многозадачную операционную среду, например Windows, то другие задачи также могут захватывать участки динамической памяти параллельно с вашей программой. Таким образом, между двумя запрошенными или высвобожденными вашей программой участками динамической памяти может вклиниться другой процесс.

Вам необходимо следить за объемом выделяемой вашей программе динамической памяти. Вы можете делать это с помощью переменных-указателей. Например, если вам нужно выделить в динамической памяти место для 20 целочисленных переменных, то вы используете для этого целочисленный указатель. Если же вам нужно выделить память для 400 чисел с плавающей точкой, то вы должны воспользоваться указателем типа `float`. Указатель всегда указывает на первое значение, хранящееся в только что выделенном участке динамической памяти. Если вы хотите получить доступ к значениям, следующим после первого значения, то вы можете воспользоваться либо нотацией указателей, либо нотацией массивов. (Видите, наше обсуждение указателей и массивов в предыдущей главе на самом деле становится полезным!)

Но <u>зачем</u> мне нужна динамическая память?

Прежде чем изучать механизмы выделения и высвобождения динамической памяти, вы, возможно, хотели бы узнать хотя бы несколько

доводов, для чего вам нужно вообще заморачиваться по поводу динамической памяти. Ведь, в конце концов, переменные, указатели и массивы, которые вы уже изучили к настоящему моменту, вполне подходили для сохранения данных программы.

Динамическая память не всегда заменит указатели и массивы, которые вы изучали до сих пор. Проблема в изученных вами переменных заключается в том, что вы должны заранее знать, какого типа и в каком количестве вам понадобятся переменные в дальнейшем. Запомните, вы должны объявить все переменные прежде, чем начнете их использовать. Если вы объявили массив, способный хранить 100 идентификационных номеров клиентов, а у пользователя 101 клиент, то ваша программа просто не сможет расширить массив во время выполнения. Некоторым программистам (таким, как вы) требуется изменить объявление массива и перекомпилировать программу, прежде чем массив сможет хранить большее количество значений.

При работе с динамической памятью, однако, не нужно заранее знать, какой объем памяти вам может понадобиться. Подобно аккордеону, динамическая память, используемая вашей программой, может растягиваться и сжиматься в соответствии с потребностями программы. Если вам потребуется еще 100 элементов массива для сохранения информации о новой группе клиентов, то программа сможет выделить память для сохранения информации об этой группе клиентов во время выполнения без необходимости перекомпиляции.

ВНИМАНИЕ

Авторы не пытаются обмануть вас и сказать, что эта глава содержит ответы на все появившиеся у вас вопросы. Полноценное овладение динамической памятью невозможно без практики, и на самом деле программы, которым действительно требуется динамическая память, не входят в поле зрения данной книги. Тем не менее благодаря используемой авторами методике, закончив чтение данной главы, у вас будет более четкое представление о том, как получить доступ к динамической памяти, чем вы бы получили в большинстве других книг.

Коммерческие программы, такие как электронные таблицы и текстовые процессоры, должны очень сильно опираться на использование динамической памяти. В конце концов, программист, создающий такую программу, не может знать, насколько большой будет электронная та-

блица или обрабатываемый текстовый документ. Такая программа, вероятнее всего, не запрашивает выделение по 1 байту по мере увеличения набираемого вами документа, так как выделение памяти не всегда эффективно, если выделять память всего лишь по 1 байту за раз. Более вероятно, что программа запрашивает выделение больших участков памяти по 100 или 500 байт.

Так почему же программисты просто не могут выделить очень большие массивы, которые бы хранили большие электронные таблицы или текстовые документы, вместо того чтобы мучиться с динамической памятью? Хотя бы из-за того, что память — это один из самых ценных ресурсов вашего компьютера. Если мы перемещаемся в сетевые или оконные среды, память становится еще более драгоценной. Ваши программы не могут каждый раз выделять такие большие массивы для действительно редких случаев, когда пользователю могут понадобиться большие объемы памяти, ведь тогда ваша программа использовала бы всю память — и другим заданиям просто не хватило бы ресурсов.

ПРИМЕЧАНИЕ

Динамическое распределение памяти позволяет вашей программе использовать ровно столько памяти, сколько необходимо. Когда пользователю нужно больше памяти (например, для ввода большего количества данных), ваша программа может получить дополнительно выделенную память. Когда ваш пользователь прекратит использование такого большого количества памяти (например, отчистив текстовый документ и начав создавать новый в текстовом процессоре), вы можете высвободить память, что сделает ее доступной для других заданий, которые, возможно, нуждаются в ней.

Как я могу выделить динамическую память?

Для использования динамической памяти вам нужно выучить лишь две новые функции. Функция `malloc()` (сокращение от английского «memory allocate» — выделение памяти) выделяет динамическую память, тогда как функция `free()` — ее освобождает.

СОВЕТ

Не забывайте подключать заголовочный файл `stdlib.h` ко всем программам, в которых планируется использование функций `malloc()` и `free()`.

Теперь, пожалуй, перейдем к сложному. Функция malloc() не самая дружелюбная функция и непроста в понимании новичкам. Возможно, самым лучшим началом будет рассмотрение примера использования функции malloc().

Предположим, вам необходимо написать программу для подсчета средних температурных показателей для местной службы погодного прогнозирования. Чем больше температурных показателей вводит пользователь, тем точнее будет прогноз. Вы решили, что для хранения 10 температурных показателей вы выделите 10 целочисленных переменных. Если пользователь пожелает ввести больше данных, то ваша программа выделит память еще на 10 целых чисел и так далее.

Для начала вам понадобится указатель на 10 значений, хранящихся в динамической памяти. Поскольку эти значения являются целыми числами, то вам понадобится целочисленный указатель. Вы должны объявить целочисленный указатель следующим образом:

```
int * temps; /* Указывает на первое значение из
динамической памяти */
```

Выделить место в динамической памяти для десяти целых чисел можно следующим образом:

```
temps = (int *) malloc(10 * sizeof(int)); /* Ого! */
```

Довольно большое количество кода для получения всего лишь 10 целых чисел. На самом деле понять вышеприведенную строку кода довольно просто. Функция malloc() требует одного единственного значения: количества байт памяти, которое необходимо выделить. Таким образом, если бы вам нужно было 10 байт, вы могли бы выделить их следующим образом:

```
malloc(10);
```

Проблема в том, что вышеописанной программе требовалось не 10 байт, а место для хранения 10 целых чисел. Сколько байт памяти нужно 10 целым числам? 10? 20? Ответ, что, конечно, это зависит от вашего компьютера и только функция sizeof() может знать наверняка.

Таким образом, если вы хотите, чтобы программа выделила место для 10 целых чисел, вы должны сообщить функции malloc(), что вам нужно выделить 10 наборов байт, при этом каждый набор должен быть достаточно велик для хранения одного целого числа. Именно поэтому

вышеприведенная строка кода содержала следующий вызов функции `malloc()`:

```
malloc(10 * sizeof(int))
```

Эта часть выражения сообщила функции `malloc()` выделить, или отложить, 10 смежных участков динамической памяти. Компьютер, в свою очередь, как бы возводит ограду вокруг этих 10 участков памяти таким образом, что последующие вызовы функции `malloc()` не вторгнутся в эту выделенную память. Теперь, когда вы поняли последнюю половину выражения `malloc()`, осталось совсем немного. Первая часть вызова функции `malloc()` очень проста.

Функция `malloc()` всегда проделывает следующие два шага (в случае, если на компьютере есть достаточное для удовлетворения ваших потребностей количество динамической памяти):

1. Выделяет запрошенное вами количество байтов памяти и гарантирует, что никакая другая программа не может перезаписать содержимое выделенной памяти до тех пор, пока ваша программа не высвободит выделенную ей память.
2. Присваивает указатель первому значению, хранящемуся в выделенной памяти

Рисунок 26.2 иллюстрирует результат вызова функции `malloc()` из предыдущего примера про температуру. Как вы можете видеть, динамическая память (показанная на рис. просто как куча) теперь содержит отделенную зону из 10 мест для целых чисел, а целочисленный указатель `temps` указывает на первое целое число. Последующие вызовы функции `malloc()` займут свободные участки динамической памяти, которые не будут пересекаться с уже выделенными участками для хранения 10 целых чисел.

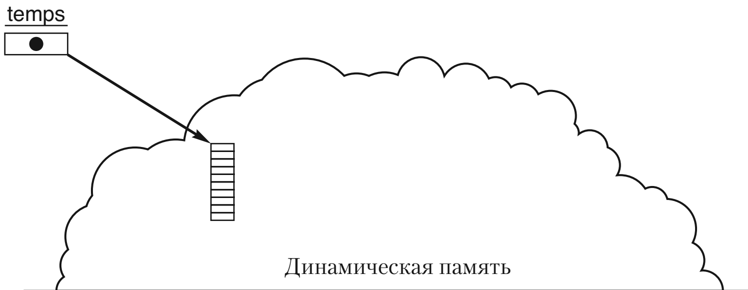


Рис. 26.2. Выделение участков памяти для 10 целых чисел

Каким образом можно использовать только что выделенные 10 ячеек памяти для хранения целых чисел? Воспринимайте их как массив! Вы можете сохранять данные, ссылаясь на ячейки как `temps[0]`, `temps[1]` и так далее. Из предыдущей главы вы уже знаете, что, используя нотацию массивов, можно получить доступ к соседствующим ячейкам памяти, даже если данный участок памяти начинается с указателя. Также помните, что ячейки каждого выделенного набора будут соседствующими, поэтому 10 целых чисел будут следовать одно за другим точно также, как если бы вы выделили участок памяти `temps` в виде целочисленного массива на 10 элементов.

Впрочем, в выделении памяти с помощью функции `malloc()` есть одна проблема. Мы еще не объяснили левую часть вызова функции `malloc()` из примера с температурами. Для чего нужно выражение `(int *)`?

Выражение `(int *)` — это приведение вида. В этой книге вы уже встречались с приведением вида, правда несколько другого типа. Чтобы привести тип `float` в тип `int` необходимо поместить выражение `(int)` перед значением типа `float`, например, следующим образом:

```
aVal = (int)salary;
```

Оператор `*` внутри выражения приведения вида означает, что перед вами выражение приведения вида указателя. Функция `malloc()` всегда возвращает символьный указатель. Если вы хотите воспользоваться функцией `malloc()` для выделения места под хранение целых чисел, чисел с плавающей точкой или данных любого другого типа, отличного от `char`, вы должны выполнить приведение вида для функции `malloc()` с тем, чтобы переменная-указатель (например, `temps`), принимающая адрес выделенного участка памяти, получила указатель правильного типа. `temps` — целочисленный указатель. Не присваивайте указателю `temps` высвобожденную функцией `malloc()` память, если вы не привели возвращаемое этой функцией значение к целочисленному указателю. Таким образом, левая часть предыдущего вызова функции `malloc()` просто сообщает функции, что на первый элемент выделенного участка памяти будет указывать не используемый по умолчанию символьный указатель, а целочисленный указатель.

ПРИМЕЧАНИЕ

Кроме определения массива в верхней части функции `main()`, какую еще выгоду вы получили от использования функции `malloc()`?

Во-первых, функцию malloc() вы можете использовать в любой части своей программы, а не только в том участке кода, где вы объявляете переменные и массивы. Таким образом, когда ваша программа готова к 100 значениям типа double, вы можете выделить эти 100 значений типа double. При использовании обычного массива, вам потребовалось бы включить выражение, подобное следующему, в верхней части кода функции main():

```
doublemyVals[100]; /* Обычный массив на 100 элементов */
```

Эти 100 значений типа double просуществуют в течение всего времени жизни программы, при этом забирая себе системные ресурсы, даже если бы эти 100 значений понадобились бы программе только на короткий период времени. Функция malloc() позволяет вам объявить только указатель, указывающий на первый элемент выделенной памяти и существующий все время жизни программы, а не целый массив.

Если недостаточно динамической памяти

В отдельных, экстремальных, случаях в системе может не быть достаточного количества памяти для удовлетворения потребностей функции malloc(). Такое может произойти, если на компьютере пользователя мало памяти, другое задание может использовать большое количество памяти или если ваша программа уже выделила под свои нужды все свободные ресурсы. Если вызов функции malloc() завершается неудачей, то переменная указатель указывает на нулевое значение, 0. Поэтому многие программисты сопровождают функцию malloc() оператором ветвления if:

```
temps = (int *) malloc(10 * sizeof(int));
if (temps == 0)
{
  printf("Упс! Недостаточно памяти! \n");
  exit(1); //Досрочное завершение программы
}
//Продолжение программы далее…
```

Если функция malloc() срабатывает нормально, то указатель temps будет содержать действительный адрес, указывающий на начало выделенного участка динамической памяти. Если вызов функции

malloc() завершается неудачей, то переменная указатель указывает на недействительный адрес 0 (адресация динамической памяти никогда не начинается с нуля), а на экран выводится сообщение об ошибке.

СОВЕТ

Программисты часто используют оператор НЕ, !, вместо сравнения значения с нулем, как было сделано в предыдущем примере. Поэтому предшествующая проверка с оператором if более вероятно будет представлена следующим образом:

```
if (!temps) /* Если значение не истинно */
```

Освобождение динамической памяти

После окончания работы с динамической памятью верните ее в систему. Чтобы сделать это воспользуйтесь функцией free(). Функцией free() намного проще пользоваться, чем функцией malloc(). Чтобы освободить память, выделенную для хранения 10 целых чисел в предшествующем вызове функции malloc() воспользуйтесь функцией free() следующим образом:

```
free(temps); /* Возвращает выделенную память в систему
*/
```

Если вы изначально выделили память для 10 значений, то все 10 ячеек памяти будут высвобождены. Если с помощью функции malloc() из динамической памяти было выделено место для хранения 1000 значений, то все 1000 ячеек будут возвращены в систему. После высвобождения памяти вы не сможете получить к ней доступ повторно. Помните, что функция free() выбрасывает выделенную память в кучу динамической памяти, где эта память может быть зачерпнута другим заданием (вспомните нашу аналогию с кучей грязи). Если вы попытаетесь воспользоваться указателем temps после применения функции free(), вы рискуете перезаписать какие-то данные, хранящиеся в памяти и, возможно, заблокировать свой компьютер, из-за чего может возникнуть необходимость перезагрузки.

Если вы забудете высвободить выделенную память, ваша операционная система попытается запросить использование этого участка памяти, однако пропуск вызова функции free() делает бессмысленным использование динамически выделяемой памяти. Цель динамической

памяти — предоставить вашей программе возможность выделить дополнительную память при возникновении необходимости и освободить эту память при завершении работы с ней.

Множественное выделение памяти

Массив указателей зачастую будет помогать вам выделять большое количество наборов участков динамической памяти. Вернемся к проблеме предсказания погоды. Предположим, метеоролог захотел ввести в программу исторические данные измерения температур сразу нескольких городов, но для каждого из этих городов количество имеющихся у него данных отличается.

Массив указателей может оказаться полезным для решения данной проблемы. Вы можете выделить массив из 50 указателей следующим образом:

```
int * temps[50]; /* 50 целочисленных указателей */
```

Данный массив (из-за оператора разыменования в объявлении) не будет содержать 50 целых чисел, перед нами массив из 50 указателей. Имя первого указателя `temps[0]`, второго — `temps[1]` и так далее. Каждый из элементов массива (каждый указатель) может указывать на отдельный набор ячеек выделенной динамической памяти. Таким образом, несмотря на то, что все элементы массива из 50 указателей должны быть определены в `main()`, вы можете выделять и освобождать данные, на которые ссылаются эти указатели, по мере потребности в дополнительной памяти.

Рассмотрим следующий участок кода, который может быть использован метеорологом:

```
for (ctr = 0; ctr < 50; ctr++)
{
puts("Сколько данных для этого города?");
scanf(" %d", &num);
//Выделение нужного количества динамической памяти
tepms[ctr] = (int *)malloc(num * sizeof(int));
//Следующий участок кода запросит ввода температур
//для каждого города
```

```
}
//Следующий участок кода, возможно, займется
вычислениями,
//связанными с температурными данными по городам
//Не забудьте высвободить динамическую память по
завершении
for (ctr = 0; ctr < 50; ctr++)
{
free(temps[ctr]);
}
```

Конечно, подобный программный код потребует объемного ввода данных. Скорее всего, данные будут передаваться из сохраненного на диске файла, а не от пользователя. Тем не менее такой код позволяет вам взглянуть на более сложные структуры организации данных, доступные при использовании динамической памяти. Кроме того, настоящие программы, как правило, — нечто большее, чем 20 строк кода, коими зачастую являются программы из этой книги. Длина настоящих программ, хотя это не говорит о том, что они сложнее представленных здесь, составляет несколько печатных страниц. По мере выполнения программы некоторым из участков кода может потребоваться дополнительная память, а другим — нет. Динамическое выделение позволяет вам использовать память более эффективно.

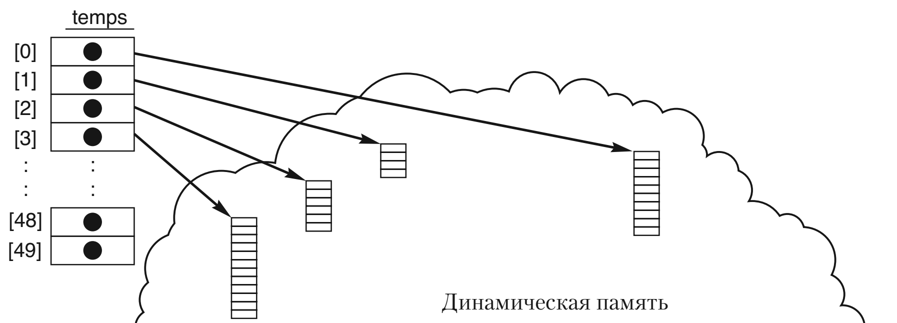


Рис. 26.3. Каждый элемент массива temps указывает на отдельный участок динамической памяти

Рисунок 26.3 — иллюстрация того, как может выглядеть «куча» динамической памяти в момент выделения участков для массива `temps` (после 4 из 50 вызовов функции `malloc()`). Как вы можете видеть,

массив `temps` находится в зоне памяти для данных программы, однако каждый элемент массива `temps` указывает на участок динамической памяти. Когда у вас исчезнет потребность в данных, на которые указывает массив `temps`, вы можете высвободить эти участки памяти.

Это была длинная глава, посвященная довольно сложной теме, но вы уже почти дочитали ее! Нам осталось лишь завершить эту главу программой, в которой использовались бы обе функции `malloc()` и `free()`, а также которая могла бы продемонстрировать вам, как написанная вами небольшая компьютерная программа может обрабатывать очень большие объемы данных.

```
//  Программа-пример №1 из главы 26
//  Руководства по C для новичков, 3-е издание
//  Файл Chapter26ex1.c
/* Эта программа запрашивает количество случайных
чисел, выделяет массив и заполняет его числами от 1
до 500, а затем проходит по всем числам и выявляет
наименьшее, наибольшее и вычисляет среднее, после чего
освобождает память. */
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
main()
{
    int i, aSize;
    int * randomNums;
    time_t t;
    double total = 0;
    int biggest, smallest;
    float average;
    srand(time(&t));
    printf("Сколько случайных чисел будет в массиве? ");
    scanf(" %d", &aSize);
    // Выделить массив, число элементов которого
    // совпадало бы с количеством случайных чисел,
    // запрошенных пользователем
```

```
    randomNums = (int *) malloc(aSize * sizeof(int));
    // Проверка правильности выделения массива
    if (!randomNums)
    {
        printf("Ошибка выделения массива случайных
чисел!\n");
        exit(1);
    }
    // Проход по элементам массива и присвоение
каждому элементу
    // целого числа в диапазоне от 1 до 500
    for (i = 0; i < aSize; i++)
    {
        randomNums[i] = (rand() % 500) + 1;
    }
    // Инициализация самого большого и самого
    // маленького числа для последующего сравнения
    biggest = 0;
    smallest = 500;
    // Проход по заполненному массиву и поиск
    // наибольшего и наименьшего чисел, а также
    // сложение всех чисел и вычисление среднего
    for (i = 0; i < aSize; i++)
    {
        total += randomNums[i];
        if (randomNums[i] > biggest)
        {
            biggest = randomNums[i];
        }
        if (randomNums[i] < smallest)
        {
            smallest = randomNums[i];
        }
    }
```

```
    average = ((float)total)/((float)aSize);
    printf("Наибольшее случайное число: %d.\n", 
biggest);
    printf("Наименьшее случайное число: %d.\n", 
smallest);
    printf("Среднее из случайных чисел: %.2f.\n", 
average);
    // При использовании malloc, не забудьте 
использовать free
    free(randomNums);
    return(0);
}
```

В данной программе взаимодействие с пользователем сведено к минимуму, от пользователя требуется ввести только нужное количество случайных чисел. Отличным вариантом проверки количества памяти на компьютере будет значительное увеличение размера массива случайных чисел. Автор данной книги смог создать массив из 12 миллионов элементов, при этом участок кода, обрабатывающий ошибку функции `malloc`, не был выполнен ни разу. На самом деле изначально при написании программы первой выдала ошибку переменная `total`, а не функция `malloc`. Изначально переменная `total` была типа `int`, и когда автор объявил массив из 10 миллионов элементов, общая сумма всех случайных чисел превысила максимально разрешенное значение переменной типа `int`, поэтому подсчет среднего значения был неверен, в конце концов, как среднее значение чисел от 1 до 500 может равняться −167! Когда переменная была увеличена до `double`, автор смог объявить и гораздо большие массивы случайных чисел.

Еще один интересный факт: чем меньше количество элементов массива, тем больше вероятность колебания значений наибольшего, наименьшего и среднего чисел. Однако чем больше количество элементов в массиве, тем больше вероятность, что наименьшее число будет равняться 1, наибольшее — 500, а среднее будет где-то посреди.

Абсолютный минимум

Функция `malloc()` выделяет вашей программе динамическую память. Доступ к динамической памяти можно получить с помощью

переменной-указателя, после чего вы можете получить доступ к остальным участкам выделенной памяти с помощью нотации массивов на основе указателя, присвоенному функции `malloc()`. По завершении работы с динамической памятью высвободите ее с помощью функции `free()`. Функция `free()` возвращает динамическую память в «кучу», чтобы этой памятью могли воспользоваться другие задания. Далее приведены ключевые концепции данной главы.

- Для выделения и высвобождения динамической памяти воспользуйтесь функциями `malloc()` и `free()`.
- Сообщите функции `malloc()`, какое количество памяти необходимо выделить с помощью оператора `sizeof()` внутри скобок функции `malloc()`.
- В начале функций объявляйте только переменные-указатели и другие переменные. Записывайте данные в динамическую память только когда вам нужны данные, отличные от простых счетчиков цикла и сумм.
- Если вам нужно следить за несколькими участками динамической памяти, воспользуйтесь массивом указателей. Каждый элемент такого массива может указывать на разное количество элементов динамической памяти.
- Проверяйте, что вызов функции `malloc()` сработал корректно. Функция `malloc()` возвращает `0` в случае возникновения ошибки.
- Не всегда следует полагаться на обычные массивы для хранения данных программы. Иногда программе нужны данные только на непродолжительный период времени, при этом использование динамической памяти повышает эффективность использования ресурсов памяти в целом.

Глава 27

УПОРЯДОЧЕНИЕ ДАННЫХ С ПОМОЩЬЮ СТРУКТУР

В этой главе

- Объявление структуры
- Запись данных в переменные-члены структуры

Массивы и указатели хорошо подходят для хранения списков значений, но эти значения должны быть одного типа. Иногда вам необходимо хранить вместе и обрабатывать как единое целое данные разного типа.

Идеальным примером такого случая может служить запись информации о клиентах. Для каждого клиента вы должны отслеживать имя (символьный массив), баланс (числа с плавающей точкой двойной точности), адрес (символьный массив), город (символьный массив), штат (символьный массив), а также почтовый индекс (символьный массив или длинное целое). Хотя вам и понадобилось бы инициализировать и распечатывать отдельные графы информации о клиенте, при записи такой информации на дисковый файл (описывается в следующей главе), вам потребовалось бы получить доступ к информации о клиенте как к единому целому.

Структура в языке С — это то средство, с помощью которого вы можете группировать данные, например такие, как информация о клиенте *и* получать доступ к каждой отдельной части этих данных, называемой *членами*. Если у вас много клиентов, и информация подобного рода встречается не единожды, то вам необходимо создать массив структур.

ПРИМЕЧАНИЕ

В других языках программирования также есть эквивалентные группировки данных, называемые *записями* (record). Однако разработчики языка С решили назвать такую группировку данных *структурой*, вот что такое «структура» в языке С.

Во многих случаях структуры содержат данные, которые вы храните на картотечных карточках размером 3×5. До появления персональных компьютеров у различных фирм был свой картотечный шкаф, в котором хранились карточки с именами клиентов, их балансами, адресами, городами и штатами проживания, а также почтовыми индексами, точно такими же как структура, которую мы только что обсудили. Позже в этой главе вы узнаете, как структуры языка C хранятся в памяти компьютера — и это позволит вам увидеть еще больше сходств с картотечными карточками.

Объявление структуры

Первое, что вам нужно сделать — сообщить компьютеру, как именно будет выглядеть ваша структура. При объявлении переменных встроенных типов данных, например `int`, у вас нет необходимости рассказывать компилятору, что такое `int` — он это уже знает. Однако при объявлении структуры вы должны сообщить компилятору, как именно будет выглядеть ваша структура. Только после этого вы сможете объявлять переменные типа данных, соответствующего вашей структуре.

Постарайтесь посмотреть на структуру просто как на группу переменных разного типа. У структуры есть имя, и ее можно рассматривать как единое значение (например, клиент), взятое в целом. Отдельные члены структуры представляют собой данные встроенных типов, например, массивы `int` или `char`, которые могут представлять возраст и имя клиента. При необходимости вы можете получить доступ к каждому члену структуры.

Структура не только похожа на картотечную карточку, но вы также можете воспринимать структуру как бумажный формуляр с полями, подлежащими заполнению. Формуляр или бланк наподобие того, что вы заполняете при подаче заявления на выпуск кредитной карты, сам по себе вещь бесполезная. Если фирма, выпускающая кредитные карты, распечатает 10 тысяч анкет, это совсем не будет значить, что у этой фирмы есть 10 тысяч клиентов. Клиент появляется только тогда, когда кто-либо заполнит такой бланк, аналогично этому, компилятор выделяет место в памяти для хранения структурной переменной только после того, как вы объявите переменную для описанной структуры.

Чтобы объявить переменную типа `int`, вам нужно сделать лишь следующее:

```
int i;
```

Перед объявлением такой переменной вам не нужно рассказывать компилятору, что такое `int`. Чтобы объявить структурную переменную, вы сначала должны объявить, как будет выглядеть структура и зарезервировать для нее имя типа данных, например `customer`. После определения формата структуры вы можете объявить переменную.

Выражение `struct` определяет вид (или макет) структуры. Далее приведен формат выражения `struct`:

```
struct [тэг структуры] {
объявление члена;
объявление члена;
…
объявление члена;
};
```

Повторимся, выражение `struct` только лишь определяет макет, или *внешний вид*, структуры. *`тэг структуры`* — это имя, которое вы даете данному конкретному макету структуры, однако *`тэг структуры`* может не иметь ничего общего с именем структурной переменной, которую вы создадите впоследствии. После объявления формата структуры вы можете начать объявлять переменные.

`объявление члена` — не что иное, как объявление переменных встроенных типов данных, как, например, `int age` из предыдущего примера. Однако в данном случае вы объявляете не переменные, а *члены*, давая имя этой части структуры.

ВНИМАНИЕ

Хотя вы можете объявить переменную одновременно с выводом объявления структуры, однако большинство программистов, работающих на языке C, так не делают. Если вы хотите объявить переменную для структуры одновременно с объявлением формата структуры, вставьте имя одной или нескольких переменных прямо перед закрывающим символом «точка с запятой» выражения `struct`.

Структуры — довольно сложная и насыщенная новой информацией тема. Следующий пример поможет вам ее понять.

Давайте представим, что вы пишете программу, которая позволяла бы отслеживать состояние складских запасов небольшой фирмы по прода-

же компьютеров. Вам нужно отслеживать наименование производителя компьютера, объем жесткого диска (в мегабайтах), объем оперативной памяти (в мегабайтах), количество, себестоимость и продажную цену.

В первую очередь, вы должны воспользоваться выражением `struct`, чтобы объявить структуру. Вот неплохой вариант:

```
struct invStruct {
char manuf[25]; //Наименование производителя
char model[15]; //Код модели
int diskSpace; //Объем НЖМД в гигабайтах
int memSpace; //Объем ОЗУ в гигабайтах
int ports; //Кол-во портов USB
int quantity; //Количество единиц на складе
float cost ; //Себестоимость компьютера
float price ; //Продажная цена компьютера
};
```

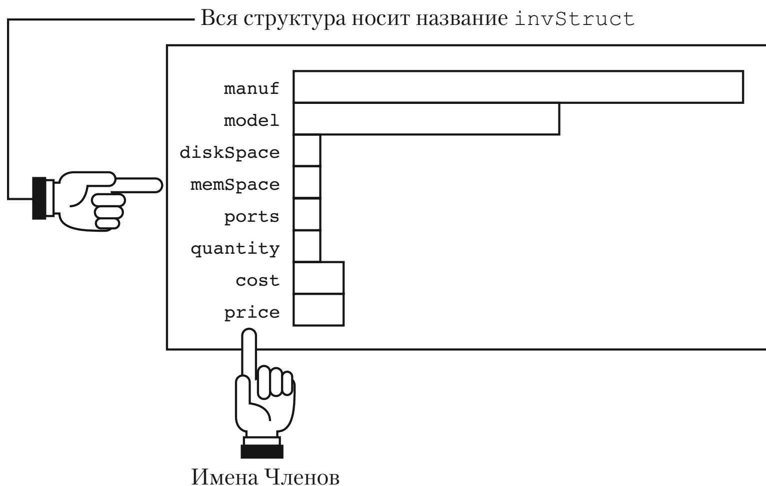


Рис. 27.1. Формат структуры `invStruct`

Предшествующее определение структуры *не содержит* объявление восьми переменных! Предшествующее определение структуры объявляет единственный структурный тип данных. Запомните, что перед определением целочисленной переменной вам не нужно сообщать компилятору, как выглядит формат `int`, однако вы *должны* рассказать

компилятору, как выглядит тип `invStruct` прежде, чем объявлять переменные этого (структурного) типа данных. Предшествующее выражение `struct` сообщает компьютеру, как должны выглядеть объявляемые пользователем переменные типа `invStruct`. После того как компилятор изучил формат структуры, компилятор может определять переменные, принимающие формат данной структуры как только пользователь будет готов эти переменные объявить.

Если вы создаете структуру, которую сможете использовать в дальнейшем повторно, поместите ее в отдельный заголовочный файл или в заголовочный файл вместе с другими часто используемыми структурами. Используйте директиву `#include` чтобы подключить такой заголовочный файл к любому исходному коду, которому может потребоваться данный файл. Если в дальнейшем вам потребуется изменить определение структуры, для этого вам потребуется найти ее только в одном месте: в заголовочном файле.

Часто программисты помещают объявления структур, такие как объявление структуры `invStruct` из предыдущего примера перед функцией `main()`, а затем объявляют переменные этого типа уже в теле функции `main()` и в телах любых других функций, следующих после функции `main()`. Чтобы создать переменные для работы со структурой, вам нужно проделать те же операции, что и при создании переменных любого другого типа: поместите имя структуры перед списком переменных. Так как в языке С нет такого типа данных, как `invStruct`, вы должны сообщить компилятору, что `invStruct` — это имя структуры. Вы можете определить три структурных переменных следующим образом:

```
#include "c:\cprogramming files\inv.h"
main()
{
struct invStruct item1, item2, item3;
// Остальной код программы...
```

Теперь вы можете записать в эти три переменные требуемые данные. Перед вами структурные переменные с именами `item1`, `item2` и `item3`. Если вам нужно было бы объявить 500 переменных, вы могли бы воспользоваться массивом:

```
#include "c:\cprogramming files\inv.h"
main()
```

```
{
struct invStruct items[500];
// Остальной код программы…
```

Помните, что, если вы пойдете этой дорогой, определение структуры должно находиться в заголовочном файле **inv.h**. В противном случае вы должны поместить определение структуры внутри программы и перед структурными переменными, например следующим образом:

```
struct invStruct {
char manuf[25]; //Наименование производителя
char model[15]; //Код модели
int diskSpace; //Объем НЖМД в гигабайтах
int memSpace; //Объем ОЗУ в гигабайтах
int ports; //Кол-во портов USB
int quantity; //Количество единиц на складе
float cost ; //Себестоимость компьютера
float price ; //Продажная цена компьютера
};
main()
{
struct invStruct items[500];
// Остальной код программы…
```

Если определение структуры `invStruct` появляется в программе перед функцией `main()`, вы можете объявлять переменные типа `invStruct` на протяжении всей программы, во всех создаваемых вами функциях. (В последней части этой книги мы расскажем вам, как писать программы, содержащие несколько функций, а не только `main()`.)

Возможно, вам могут понадобиться три указателя на структуру, а не просто структурные переменные. В этом случае объявите их следующим образом:

```
main()
{
struct invStruct *item1, *item2, *item3;
// Остальной код программы…
```

Теперь `item1`, `item2` и `item3` могут указывать на три структурные переменные. После создания таких указателей вы сможете резервировать для структуры участки динамической памяти вместо использования простых переменных. (Функция `sizeof()` также работает со структурными переменными, что делает возможным записывать в динамическую память структурные данные.) Следующие три выражения резервируют три участка динамической памяти для хранения структуры и привязывают указатели `item1`, `item2` и `item3` к этим участкам памяти:

```
item1 = (struct invStruct *)malloc(sizeof(invStruct));
item2 = (struct invStruct *)malloc(sizeof(invStruct));
item3 = (struct invStruct *)malloc(sizeof(invStruct));
```

Запись данных в структурные переменные

Новый оператор, *оператор-точка*, позволяет записать данные в какой-то конкретный член структурной переменной. Ниже приведен формат оператора-точки:

```
имяСтруктурнойПеременной.имяЧлена
```

Слева от точки всегда должно находиться имя структурной переменной, например, `item1` или `employee[16]`. Справа от точки всегда находится имя какого-либо члена структуры, например `quantity`, `cost` или `name`. Оператор-точка может записать данные только в поименованные структурные переменные. Если вы хотите записать данные в структуру, хранящуюся в динамической памяти и на которую ссылается указатель на структурную переменную, то вам нужно воспользоваться *оператором структурного указателя*, `->`.

В следующей программе объявляется массив из трех структурных переменных, при этом используется тег структуры `bookInfo`, определенной в файле `bookInfo.h`, приведенном в самом начале. Пользователя просят заполнить структурные переменные, а затем программа указывает на них. В следующих двух главах вы научитесь выводить данные структурных переменных в дисковые файлы для долгосрочного хранения.

Первый файл — это заголовочный файл, содержащий определение структуры:

```
// Программа-пример №А из главы 27
// Руководства по C для новичков, 3-е издание
// Файл bookinfo.h
// В этом заголовочном файле объявляется структура для
записи
// информации о книге
struct bookInfo {
    char title[40];
    char author[25];
    float price;
    int pages;
};
```

А теперь — файл программы **.c**:

```
// Программа-пример №1 из главы 27
// Руководства по C для новичков, 3-е издание
// Файл Chapter27ex1.c
/* Подключив файл bookInfo.h, данная программа
получает
структуру bookInfo, просит пользователя заполнить три
структуры, а затем распечатывает их.
*/
//В первую очередь, подключим файл с определением
структуры
#include "bookinfo.h"
#include <stdio.h>
main()
{
    int ctr;
    struct bookInfo books[3]; // Массив из трех
структурных переменных
    // Получить информацию о каждой книге от
пользователя
    for (ctr = 0; ctr < 3; ctr++)
    {
```

```
        printf("Введите название книги №%d?\n",
(ctr+1));
        gets(books[ctr].title);
        puts("Кто автор? ");
        gets(books[ctr].author);
        puts("Сколько стоила книга? ");
        scanf(" $%f", &books[ctr].price);
        puts("Сколько страниц в книге? ");
        scanf(" %d", &books[ctr].pages);
        getchar(); //Создает пустую строку для
следующего прохода
    }
    //Вывести заголовок, а затем с помощью цикла
распечатать
    //информацию
    printf("\n\nМоя коллекция книг: \n");
    for (ctr = 0; ctr < 3; ctr++)
    {
        printf("№%d: %s автор %s", (ctr+1),
books[ctr].title,
                books[ctr].author);
        printf("\nСодержит %d страниц и строит $%.2f",
     books[ctr].pages, books[ctr].price);
        printf("\n\n");
    }
    return(0);
}
```

Если бы вы сохранили структурные переменные в динамической памяти, то вы бы не смогли воспользоваться оператором-точкой, так как данному оператору требуется имя переменной. Для сохранения данных структуры в динамической памяти воспользуйтесь оператором ->.

Оператор -> требует имя указателя слева и имя члена структуры справа. Далее приведена программа, эквивалентная программе из предыдущего примера с той лишь разницей, что вместо переменной и оператора-точки используется динамическая память и оператор ->.

```
// Программа-пример #2 из Главы 27
// Руководства по C для новичков, 3-е издание
// Файл Chapter27ex2.c
/* Как и предыдущая, данная программа пытается
заполнить структуры некой информацией, однако в этот
раз используется массив указателей.
*/
//В первую очередь, подключим файл с определением
структуры
#include "bookinfo.h"
#include <stdio.h>
#include <stdlib.h>
main()
{
    int ctr;
    struct bookInfo *books[3]; //Массив из трех
структурных переменных
    // Получить информацию о каждой книге от
пользователя
    for (ctr = 0; ctr < 3; ctr++)
    {
        books[ctr] = (struct bookInfo*)
malloc(sizeof(struct bookInfo));
       printf("Введите название книги №%d?\n",
(ctr+1));
        gets(books[ctr]->title);
        puts("Кто автор? ");
        gets(books[ctr]->author);
        puts("Сколько стоила книга? ");
        scanf(" $%f", &books[ctr]->price);
        puts("Сколько страниц в книге? ");
        scanf(" %d", &books[ctr]->pages);
        getchar(); //Создает пустую строку для
следующего прохода
    }
```

```
    //Вывести заголовок, а затем с помощью цикла
распечатать
    //информацию
    printf("\n\nМоя коллекция книг: \n");
    for (ctr = 0; ctr < 3; ctr++)
    {
        printf("№%d: %s автор %s", (ctr+1),
books[ctr]->title,
                books[ctr].author);
        printf("\nСодержит %d страниц и строит $%.2f",
books[ctr]
->pages, books[ctr]->price);
        printf("\n\n");
    }
    return(0);
}
```

Абсолютный минимум

Целью данной главы было ознакомить вас со структурами. *Структура* — это агрегированный переменный тип данных. В то время как массив должен содержать значения одного и того же типа, структура может содержать значения разного типа данных.

Прежде чем использовать структурную переменную, вы должны с помощью выражения `struct` сообщить компилятору, как именно будет выглядеть созданная вами структура. Выражение `struct` информирует компилятор о точном количестве и типе данных каждого из членов структуры. Структура напоминает группу из одной или нескольких переменных разного типа. Ниже приведены основные концепции данной главы.

- При необходимости сгруппировать элементы различного типа данных, воспользуйтесь структурой.
- Определите структуру прежде, чем объявлять структурные переменные.
- Для получения доступа к нужным членам структурной переменной воспользуйтесь оператором-точкой.

- Для получения доступа к нужным членам структурной переменной, на которую ссылается переменная-указатель, воспользуйтесь оператором -> (*оператор структурного указателя*).
- Не используйте имена членов структуры в качестве переменных. Имена членов существуют только для того, чтобы у вас была возможность работать с нужными вам отдельными частями структуры.
- Не забывайте вставлять точку с запятой в конце каждого определения структуры.
- Не путайте оператор-точку и оператор структурного указателя. Помните, что имя структурной переменной должно указываться перед оператором-точкой, а перед оператором -> необходимо вводить имя структурной переменной-*указателя*.

Глава 28
СОХРАНЕНИЕ ПОСЛЕДОВАТЕЛЬНЫХ ФАЙЛОВ НА КОМПЬЮТЕРЕ

В этой главе

- Запись информации в файлы на диске
- Открытие файла
- Использование последовательных файлов

Ни одна из рассмотренных нами до настоящего момента программ не могла отправлять данные на длительное хранение. Представьте на секунду: если вы объявили целочисленную переменную и присвоили ей значение `14`, а затем выключили компьютер (сейчас примите это на веру, а позже можете проверить), то в переменной это значение уже сохранено не будет. Если бы вы затем опять включили компьютер и попытались отыскать данное значение в переменной — вы бы его не нашли: на это нет никаких шансов.

В этой главе мы расскажем вам, как сохранять данные на диск. Когда данные находятся на диске, они там до тех пор, пока вы их не измените или не сотрете. Данные на диске подобны музыке на кассете: вы можете выключить магнитофон, но музыка останется на кассете, пока вы ее не перезапишете. На самом деле, совершенно нет причин заставлять пользователя многократно вводить одни и те же данные, например журнал уровня продаж.

ПРИМЕЧАНИЕ

Файлы очень важны для компьютерных программ, обрабатывающих данные. Какая польза была бы от текстовых редакторов, если бы они не умели работать с файлами?

Файлы на диске

Данные на дисках хранятся в *файлах*. Если вы когда-либо сохраняли программу на языке C на диске, то вы уже имеете представление о кон-

цепции файлов. Они могут содержать либо программы, либо данные. Прежде, чем вы сможете запустить программу, она должна быть сначала перенесена с диска в оперативную память компьютера. Кроме того, прежде, чем вы сможете работать с данными, вы должны будете их записать в переменные. Переменные также хранят данные до их записи на дисковый файл.

Существует два типа файлов: *последовательные файлы* (файлы последовательного доступа) и *файлы произвольного доступа*. Данные типы файлов говорят о способе доступа к данным. Если вы работаете с последовательным файлом, то вы должны считывать и записывать данные в порядке их следования. В случае с файлами произвольного доступа, вы можете перемещаться по файлу, считывая и записывая данные в разных участках этого файла.

СОВЕТ

Последовательный похож на видеокассету, а файл произвольного доступа — на диск DVD или Blu-Ray. В случае с кассетой вы вынуждены смотреть фильм в том порядке, как он записан на кассету (либо перематывать его вперед/назад, но также в порядке записи), тогда как диски DVD и Blu-Ray позволяют произвольно переключаться между разными разделами и главами.

Открытие файла

Чтобы открыть файл вам нужно воспользоваться функцией `fopen()`, описание которой находится в том же заголовочном файле, что и описание функции `printf()`: `stdio.h`. Прежде, чем рассматривать функцию `fopen()` в действии, вы должны получить представлении об указателе на файл.

ПРИМЕЧАНИЕ

В концепции указателя на файл очень легко разобраться. Обычный указатель содержит адрес данных, записанных в переменную. Указатель на файл содержит местоположение требуемого файла на диске.

Чтобы объявить указатель на файл необходимо ввести специальное выражение. Как и в случае с простой переменной, вы можете присвоить

указателю на файл любое имя. Предположим, вам нужно открыть файл с информацией о работнике. Перед использованием функции `fopen()` вы должны объявить переменную-указатель на файл. Предположим, что вы назвали такой указатель `ftpr`, тогда его объявление будет выглядеть следующим образом:

```
FILE * fptr; /* Объявление указателя на файл с именем
ftpr */
```

ВНИМАНИЕ

Большинство программистов, работающих на языке C, объявляют указатели на файлы перед функцией `main()`. Это позволяет сделать такой указатель *глобальным*, что означает, что данным указателем можно воспользоваться во всех блоках программы. (Большая часть переменных *локальные*, а не глобальные.) Так как часть выражения, объявляющего указатель на файл, написана заглавными буквами, это означает, что `FILE` определено где-то с помощью директивы `#define`. На самом деле, `FILE` определяется в заголовочном файле `stdio.h`, в чем заключается самая главная причина, почему вы должны включить файл `stdio.h` в свою программу, использующую данные, хранящиеся в файле на диске.

После объявления указателя, вы можете ассоциировать данный указатель с нужным файлом с помощью функции `fopen()`. После осуществления вызова функции `fopen()` вы можете использовать затребованный файл по всему тексту программы. Например, файл **C:\cprograms\cdata.txt** может быть открыт следующим образом.

СОВЕТ

Если на вашем компьютере нет диска **C:**, замените `C:` из данного примера на любую другую букву диска. Если вы хотите поместить свои файлы в какую-то конкретную папку, но не знаете путь к этой папке, щелкните правой кнопкой мыши по любому файлу, находящемуся в этой папке, и из открывшегося контекстного меню выберите пункт **Properties** (Свойства). В открывшемся диалоговом окне вы сможете просмотреть путь к нужной папке, вы можете использовать этот путь для вызова функции `fopen()`.

```
#include <stdio.h>
FILE *fptr; //Объявление указателя на файл
```

```
main()
{
fptr = fopen("c:\cprograms\cdata.txt", "w");
//далее текст программы
fclose (fptr); //Всегда закрывайте открытые файлы
```

По тексту программы вы будете получать доступ к файлу **cdata.txt** не по имени, а с помощью указателя. Использовать переменную-указатель на файл проще, чем каждый раз прописывать имя файла с указанием полного пути каждый раз, когда требуется получить доступ к файлу, кроме того, указатель позволяет избежать опечаток.

ВНИМАНИЕ

После окончания работы с архивом или каталогом не забудьте задвинуть ящики обратно, иначе вы можете больно удариться головой! Закрывайте все открытые ранее файлы по окончание работы с ними, в противном случае часть данных может быть утеряна. Функция `fclose()` — антоним функции `fopen()`. В скобках функции `fclose()` необходимо указать имя указателя на файл, который вы хотите закрыть.

Если значение указателя на файл равняется `0`, то вы знаете, что произошла ошибка. Вызов функции `fopen()` возвращает `0`, если произошла ошибка открытия файла. Например, нуль будет возвращен, если вы попытаетесь открыть файл на несуществующем томе диска.

`"w"` — второй аргумент вызова функции `fopen()` из предыдущего примера — сокращение от английского слова «*write*» — «записать». Вторым аргументом функции `fopen()` должен быть один из строковых *модификаторов*, приведенных в табл. 28.1.

Табл. 28.1. Основные строковые модификаторы функции `fopen()`

Модификатор	**Описание**
"w"	Режим записи, создающий новый файл, вне зависимости от того, существует ли уже файл с указанным именем
"r"	Режим чтения, позволяющий прочитать существующий файл. Если файл не существует, программа выдаст ошибку
"a"	Режим добавления (дозаписи), позволяющий дозаписать данные в конец файла или создать новый файл, если файл с указанным именем не существует.

Использование файлов последовательного доступа

С файлами последовательного доступа вы будете выполнять только три операции: создание, чтение и добавление информации (запись). Для выполнения записи в файл вы можете воспользоваться функцией `fprintf()`. Освоить функцию `fprintf()` легко, так как она является аналогом функции `printf()`, с той лишь разницей, что первым аргументом данной функции должен быть указатель на файл. Следующая программа создает файл и записывает в него с помощью функции `fprintf()` некоторые данные:

```
// Программа-пример №1 из главы 28
// Руководства по С для новичков, 3-е издание
// Файл Chapter28ex1.c
/* В основе этой программы программа из Главы 27
(информация о книгах), но сведения о книгах печатаются
в файл bookinfo.txt. */
//В первую очередь подключить файл с определением
структуры
#include "bookinfo.h"
#include <stdio.h>
#include <stdlib.h>
FILE * fptr;
main()
{
    int ctr;
    struct bookInfo books[3]; // Массив из трех
    структурных переменных
    // Получить информацию о каждой книге от
    пользователя
    for (ctr = 0; ctr < 3; ctr++)
    {
        printf("Введите название книги №%d?\n",
(ctr+1));
        gets(books[ctr].title);
        puts("Кто автор? ");
```

```
        gets(books[ctr].author);
        puts("Сколько стоила книга? ");
        scanf(" $%f", &books[ctr].price);
        puts("Сколько страниц в книге? ");
        scanf(" %d", &books[ctr].pages);
        getchar(); //Создает пустую строку для
        следующего прохода
    }
    // При вводе пути к файлу не забудьте
    // продублировать обратную
    // косую черту, иначе компилятор воспримет ее как
    // начало символа преобразования
    fptr = fopen("C:\\users\\DeanWork\\Documents\\
BookInfo.txt","w");
    // Проверить, открылся ли файл
    if (fptr == 0)
    {
        printf("Ошибка--Невозможно открыть файл.\n");
        exit (1);
    }
    // Напечатать файл заголовок, а затем пройти
    // циклом по массиву и распечатать информацию
    // о книгах, но не на экране, а в файл
    fprintf(fptr, "\n\Моя коллекция книг: \n");
    for (ctr = 0; ctr < 3; ctr++)
    {
        fprintf(fptr, "№%d: %s автор %s", (ctr+1),
books[ctr].title,
                books[ctr].author);
        fprintf(fptr, "\nСодержит %d страниц и стоит
$%.2f",
                books[ctr].pages, books[ctr].price);
        fprintf(fptr, "\n\n");
    }
```

```
    fclose(fptr); // Всегда закрывайте файлы
    return(0);
}
```

Если вы запустили программу и заблокировали содержимое файла **bookinfo.txt** (просто найдите этот файл и дважды щелкните по нему мышью, Блокнот не должен его открыть). На экране вы увидите введенную вами информацию о книгах, на нашем экране сообщение выглядело следующим образом:

```
Моя коллекция книг:
№1: Count Trivia автор Дин Миллер
Содержит 250 страниц и стоит: $14.99
№2: Moving from C to C++ автор Грэг Перри
Содержит 600 страниц и стоит $39.99
№3: Противостояние автор Стивен Кинг
содержит 1200 страниц и стоит $24.99
```

Миллер, Перри и Кинг: как приятно видеть этих трех замечательных современных авторов, собранных в одном файле! Плюсом повторного использования программы из главы 27 в том, что оно позволяет наглядно проиллюстрировать, как легко вы можете применить новые знания на базе уже изученного материала (и уже написанных программ) и организовать работу с файлами. Все, что нам потребовалось — это объявить указатель на файл, открыть нужный файл (и удостовериться в том, что файл был открыт правильно), а также заменить все вызовы функции `printf()` на функцию `fprintf()` для записи в файл информации, ранее выводимой на экран.

ВНИМАНИЕ

Открытие файла в режиме `"w"` перезапишет существующий файл с таким же именем. Таким образом, если вы дважды запустите предыдущую программу, в файле будут отображены только данные, введенные вами во время последнего запуска. Если вы хотите добавить данные в файл, сохранив уже имеющиеся, откройте файл в режиме дозаписи `"a"`.

Теперь, научившись записывать данные файл, не хотите ли приступить к изучению способов получения информации из файлов? Для чте-

ния содержимого файла воспользуйтесь функцией `fgets()`. Функция `fgets()` — аналог функции `gets()`, который можно переадресовать на дисковый файл. Функция `fgets()` считывает строки текста из файла в символьные массивы (или выделенный участок динамической памяти, ссылкой на который является символьный указатель).

ПРИМЕЧАНИЕ

Воспринимайте префикс *f* в названиях функций `fputs()` `fprintf()`? и `fgets()` как сокращение от английского слова «*file*» — «файл». Функции `puts()` и `gets()` работают с экраном и клавиатурой соответственно, тогда как функции `fputs()` и `fgets()` записывают и считывают данные в файл или из файла.

В отличие от функции `gets()`, функция `fgets()` требует указания максимальной длины массива, в который производится считывание данных. Если вы невнимательны, то можете случайно прочитать информацию после конца файла, что приведет к возникновению ошибки. Не забывайте проверять положение конца файла.

Функция `fgets()` считывает информацию построчно. Если вы дадите функции задание прочитать символов больше, чем находится в строке, функция `fgets()` автоматически прекратит считывание данных, но только в том случае, если строка текста завершается символом новой строки.

Предыдущая программа, которая создала файл **bookInfo.txt**, всегда прописывала символ `\n` в конце каждой строки, таким образом, что последующие функции `fgets()` могут произвести построчное чтение файла.

Следующая программа проходит циклом по файлу (в данном случае это файл bookInfo.txt, созданный программой из предыдущего примера) и выводит считанную информацию на экран.

```
// Программа-пример №2 из главы 28
// Руководства по C для новичков, 3-е издание
// Файл Chapter28ex2.c
/* Эта программа использует файл информации о книгах
из первой программы главы 28, считывает каждую строку
и выводит информацию на экран. */
#include <stdio.h>
```

```
#include <stdlib.h>
FILE * fptr;
main()
{
    char fileLine[100]; // Будет хранить вводимые
строки
    fptr = fopen("C:\\users\\DeanWork\\Documents\\
BookInfo.txt","r");
    if (fptr != 0)
    {
        while (!feof(fptr))
        {
            fgets(fileLine, 100, fptr);
            if (!feof(fptr))
            {
                puts(fileLine);
            }
        }
    }
    else
    {
    printf("\nОшибка при открытии файла.\n");
    }
    fclose(fptr); // Всегда закрывайте файлы
    return(0);
```

Функция `feof()` возвращает ИСТИНА, если программой была прочитана последняя строка текста из файла. В предыдущей программе в функции `feof()` нет большой необходимости, так как мы точно знаем, сколько строк текста содержится в файле **bookInfo.txt**. (Мы сами создали этот файл с помощью предыдущей программы.) В данном случае мы знаем, сколько строк текста содержится в файле, однако вы должны все время использовать данную функцию при чтении информации из файлов на диске, ведь зачастую вы не можете знать, сколько именно строк текста есть в требующемся файле, так как в этот файл могли быть внесены изменения с помощью других программ.

ВНИМАНИЕ

В вызове функции `fprintf()` указатель на файл является первым аргументом, тогда как в вызове функции `fgets()` — *последним*. Нет ничего важнее правильной последовательности аргументов!

Для считывания отдельных числовых значений вы также можете воспользоваться функцией `fscanf()`, если вы записали эти значения с помощью соответствующих функций `frpintf()`.

Вы можете добавить данные в файл, открыв этот файл в режиме дозаписи — и вывести в него нужные данные из программы. Следующая программа добавляет строку "`Скоро я куплю еще книг`" в конец файла данных **bookInfo.txt**:

```
// Программа-пример №3 из главы 28
// Руководства по C для новичков, 3-е издание
// Файл Chapter28ex3.c
/* Эта программа открывает существующий файл
информации о книгах из первого примера главы 28
и добавляет в конец этого файла одну строку. */
#include <stdio.h>
#include <stdlib.h>
FILE * fptr;
main()
{
    fptr = fopen("C:\\users\\DeanWork\\Documents\\
BookInfo.txt","r");
    if (fptr == 0)
    {
        printf("Ошибка открытия файла! Извините!\n");
        exit (1);
    }
    // Прибавить строку в конце
    fprintf(fptr, "\nСкоро я куплю еще книг!\n");
    fclose(fptr); // Всегда закрывайте файлы
    return(0);
}
```

Взгляните теперь на содержимое файла **bookInfo.txt** (обратите внимание на новую строку):

```
Моя коллекция книг:
№1: Count Trivia автор Дин Миллер
Содержит 250 страниц и стоит: $14.99
№2: Moving from C to C++ автор Грэг Перри
Содержит 600 страниц и стоит $39.99
№3: Противостояние автор Стивен Кинг
Содержит 1200 страниц и стоит $24.99
Скоро я куплю еще книг!
```

Абсолютный минимум

Целью данной главы было показать вам как создавать, читать и записывать файлы последовательного доступа. Прежде чем данные могут быть прочитаны или записаны, ваша программа должна открыть соответствующий файл. После завершения работы с файлом программа должна его закрыть.

При чтении из файла проверяйте, не достигли ли вы конца файла, и убедитесь, что не пытаетесь прочесть информацию после конца файла. Функция `feof()` является встроенной функцией языка С, она позволяет вам проверить, не достигли ли вы конца файла. Далее приведены основные концепции данной главы.

- Сохраняйте данные длительного хранения в файлах.
- Перед использованием файла откройте его с помощью функции `fopen()`.
- По завершении работы всегда закрывайте файл с помощью функции `fclose()`.
- Не производите чтение файла без использования функции `feof()`, так как, возможно, вы уже прочитали последнюю строку файла
- Не забывайте, что указатель на файл является *первым* аргументом функции `fprintf()` и *последним* аргументом функции `fgets()`.

Глава 29

СОХРАНЕНИЕ ФАЙЛОВ ПРОИЗВОЛЬНОГО ДОСТУПА НА КОМПЬЮТЕРЕ

В этой главе

- Открытие файлов случайного доступа
- Перемещение по файлу

В этой главе мы покажем вам, как перемещаться от одного участка файла к другому, читать и записывать данные по мере такого перемещения. В предшествующей главе мы представили вам методы, которые вы можете использовать для записи, чтения или добавления информации в файл. Проблема заключается лишь в том, что, если вы открыли файл последовательного доступа для чтения, вы можете *только* читать этот файл.

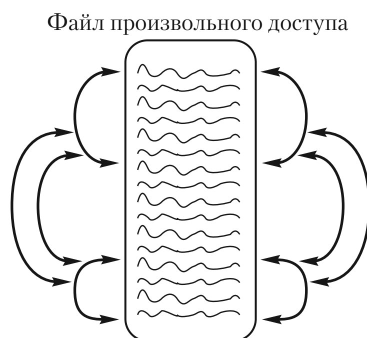


Рис. 29.1. Файлы произвольного доступа позволяют считывать и записывать информацию в любом порядке

Иногда вам может понадобиться прочитать информацию о клиенте с диска и изменить баланс этого клиента. Разумеется, вы вряд ли захотите создавать новый файл только для записи этого небольшого изменения. Вместо этого вы захотели бы считать информацию о клиен-

те в переменную, изменить ее, а затем заново записать информацию на диск точно туда, где она и была до внесения изменений. Как показано на рис. 29.1, файлы произвольного доступа позволяют перемещаться внутри файла, записывая и считывая информацию в любом участке такого файла.

Физическое размещение файла не определяет его тип (будь то файл произвольного или последовательного доступа). Вы можете создать файл последовательно, а затем читать и изменять его произвольно. Для языка С файл — это лишь поток байтов и способ доступа к нему не привязан к какому-либо формату файла.

Открытие файлов произвольного доступа

Чтобы прочитать или записать файл в режиме произвольного доступа, этот файл сначала должен быть открыт в режиме произвольного доступа. В табл. 29.1 перечислены режимы произвольного доступа к файлу. Как вы можете видеть, краеугольный камень произвольного доступа к файлам — использование уже ранее изученных вами в предыдущей модификаторов доступа с добавлением знака плюс.

Табл. 29.1. Режимы произвольного доступа к файлу с помощью `fopen()`

Модификатор	Описание
`"r+"`	Открывает существующий файл для чтения и записи
`"w+"`	Открывает новый файл для чтения и записи
`"a+"`	Открывает файл в режиме дозаписи (указатель на файл указывает на конец файла), но позволяет вернуться назад по тексту файла и совершить чтение или запись по мере необходимости

ПРИМЕЧАНИЕ

Как и в случае с последовательными файлами, режим доступа — это строковый аргумент, являющимся последним в вызове функции `fopen()`. Закрыть файл произвольного доступа можно так же, как и последовательный файл: с помощи функции `fclose()`.

Все три режима доступа позволяют вам считывать и записывать информацию в файл. Выбираемый вами режим доступа зависит от того, какую операцию над файлом вы хотели бы произвести *в первую очередь*. Если нужный файл уже существует и вы хотите получить к нему про-

извольный доступ, воспользуйтесь модификатором режима доступа `r+`. Если вы хотите создать новый файл, воспользуйтесь режимом `w+` (если файл уже существует, то C перезапишет существующую версию). Если вы хотите добавить информацию в конец файла, но также опционально «отмотать назад» и считать и/или записать какую-то информацию, воспользуйтесь режимом `a+`. Далее приведен пример вызова функции `fopen()`, открывающий новый файл для чтения и записи:

```
fptr = fopen("C:\\Users\\DeanWork\\letters.txt", "w+");
```

Как и в случае с файлами последовательного доступа, переменная `fptr` должна быть указателем на файл. Двойной обратный слеш необходим при указании пути к файлу. Помните, что функция `fopen()` возвращает нуль, если при открытии файла произошла ошибка.

СОВЕТ

Вы можете сохранить имя файла в массиве символов и использовать имя массива вместо указания действительного пути к файлу.

Перемещение по файлу

Для перемещения по файлу пользуйтесь функцией `fseek()`. После открытия файла система инициализирует указатель на файл таким образом, чтобы он указывал на следующий участок файла, который можно прочитать или в который можно записать информацию. Функция `fseek()` смещает указатель таким образом, что для записи и чтения становятся доступны те участки файла, которые были бы недоступны при последовательном доступе. Ниже приведен формат функции `fseek()`:

```
fseek(filePtr, longVal, начало);
```

`filePtr` — это указатель на файл, использованный в функции `fopen()` при открытии файла в режиме произвольного доступа. `longVal` — это переменная типа `long int` или литерал, который может быть как положительным, так и отрицательным. Переменная `longVal` представляет собой количество байтов, которое нужно пропустить (вперед или назад) при перемещении по файлу. Аргумент `начало` — это всегда одно из значений, показанных в табл. 29.2. Переменная `начало` говорит функции, откуда начинать `fseek()` поиск нужного фрагмента файла:

Табл. 29.2. Значения переменной `начало`, которые могут появиться в функции `fseek()`

начало	**Описание**
SEEK_SET	Начало файла
SEEK_CUR	Текущая позиция в файле
SEEK_END	Конец файла

Аргумент `начало` сообщает компьютеру, *с какой позиции* начать произвольный доступ к файлу. После того как вы установили позицию указателя на файл с помощью функции `fseek()`, вы можете воспользоваться функциями файлового ввода и вывода для записи и чтения информации.

Если вы установите указатель на файл в конец файла (с помощью модификатора `SEEK_END`), а затем начнете запись данных в файл, то новые данные будут добавлены в конец файла. Если же вы установите указатель на файл в участок с уже имеющимися данными (модификаторы `SEEK_SET` и `SEEK_CUR`), а затем начнете запись данных, то новые данные затрут уже имеющиеся (будут записаны «поверх» этих данных).

ВНИМАНИЕ

Используйте функцию `fseek()` только с файлами произвольного доступа. Доступ к последовательным файлам может осуществляться только в порядке следования данных.

Значения, перечисленные в табл. 29.2, записаны заглавными буквами, что наталкивает на мысль о том, что они должны быть где-то определены. Действительно, данные модификаторы объявлены в файле `stdio.h` с помощью директив `#define`.

Следующая программа открывает файл в режиме произвольного доступа, записывает в него буквы латинского алфавита от A до Z, а затем считывает эти буквы в обратном порядке.

При использовании модификатора произвольного доступа `"w+"` нет необходимости повторно открывать файл для чтения данных.

```
// Программа-пример №1 из главы 29
// Руководства по C для новичков, 3-е издание
// Файл Chapter29ex1.c
```

```
/* Эта программа открывает файл letters.txt и печатает
в него буквы от A до Z. После этого программа
с помощью цикла считывает буквы в обратном порядке от
Z до A и выводит их на экран */
#include <stdio.h>
#include <stdlib.h>
FILE * fptr;
main()
{
    char letter;
    int i;
    fptr = fopen("C:\\users\\deanwork\\documents\\
                letters.txt","w+");
    if (fptr == 0)
    {
     printf("При открытии файла произошла ошибка.\n");
        exit(1);
    }
    for (letter = 'A'; letter <= 'Z'; letter++)
    {
        fputc(letter, fptr);
    }
    puts("Завершена запись букв от A до Z");
    // Чтение файла в обратной последовательности
    fseek(fptr, -1, SEEK_END); // Минус 1 байт с конца
    printf("Данные файла в обратном порядке:\n");
    for (i = 26; i > 0; i--)
    {
        letter = fgetc(fptr);
        // Чтение буквы и возврат на 2 позиции
        fseek(fptr, -2, SEEK_CUR);
        printf("The next letter is %c.\n", letter);
    }
  fclose(fptr); // Повторимся, всегда закрывайте файлы
    return(0);
}
```

СОВЕТ

Как вы можете видеть `fputc()` — отличная функция для вывода отдельных символов из программы в файл. Функция `fgetc()` считывает отдельные символы из файла. Функции `fputc()` и `fgetc()` также соотносятся с функциями `putc()` и `getc()`, как `fputs()` и `fgets()` с `puts()` и `gets()`.

Пока вы, возможно, еще не уловили смысла в использовании файлов произвольного доступа. Произвольный доступ предоставляет вам преимущество в возможности записи данных в файл и последующего считывания записанных данных без необходимости закрывать и повторно открывать файл. Кроме того, функция `fseek()` позволяет установить указатель на файл в любую позицию на любое количество байтов относительно начала, середины или конца файла.

Предполагая, что файл с буквами, созданный в предыдущей программе, все еще находится на диске, следующая программа спрашивает пользователя, какую позицию необходимо изменить. После этого программа с помощью функции `fseek()` устанавливает указатель на файл в нужную позицию и записывает в эту позицию символ *. После этого, опять же с помощью функции `fseek()`, указатель возвращается в начало файла — и программа заново распечатывает файл.

```
// Программа-пример №1 из главы 29
// Руководства по С для новичков, 3-е издание
// Файл Chapter29ex1.c
/* Эта программа открывает файл letters.txt и печатает
в него буквы от A до Z. После этого программа
с помощью цикла считывает буквы в обратном порядке от
Z до A и выводит их на экран */
#include <stdio.h>
#include <stdlib.h>
FILE * fptr;
main()
{
    char letter;
    int i;
    fptr = fopen("C:\\users\\deanwork\\documents\\
letters.txt",
```

```
                    "w+");
    if (fptr == 0)
    {
        printf("При открытии файла произошла
ошибка.\n");
        exit(1);
    }
    printf("Букву с каким № нужно заменить (1-26)? ");
    scanf(" %d", &i);
    // Поиск указанной позиции от начала файла
    fseek(fptr, (i-1), SEEK_SET); // Вычтем 1 из
номера позиции
                                  // потому что
массивы начинаются с 0
    // Заменить букву в этой позиции на *
    fputc('*', fptr);
    // Вернуться к началу файла и распечатать его
    fseek(fptr, 0, SEEK_SET);
    printf("Так файл выглядит сейчас:\n");
    for (i = 0; i < 26; i++)
    {
        letter = fgetc(fptr);
        printf("Следующая буква %c.\n", letter);
    }
    fclose(fptr); // Повторимся, всегда закрывайте
файлы
    return(0);
}
```

Программа распечатывает содержимое файла, при этом вместо буквы, номер, которой был указан пользователем, выводится символ *. Далее приведен результат тестового запуска программы:

```
Следующая буква A.
Следующая буква B.
Следующая буква C.
```

```
Следующая буква D.
Следующая буква E.
Следующая буква F.
Следующая буква G.
Следующая буква *.
Следующая буква I.
Следующая буква J.
Следующая буква K.
Следующая буква L.
Следующая буква M.
Следующая буква N.
Следующая буква O.
Следующая буква P.
Следующая буква Q.
Следующая буква R.
Следующая буква S.
Следующая буква T.
Следующая буква U.
Следующая буква V.
Следующая буква W.
Следующая буква X.
Следующая буква Y.
Следующая буква Z.
```

Как вы можете видеть, восьмая позиция в файле с латинским алфавитом теперь содержит * вместо буквы H. Оставшаяся часть файла осталась без изменений.

ПРИМЕЧАНИЕ

Если вы запустите эту программу второй раз и измените другую букву (например, номер 15 — O), то ваш файл будет распечатан со звездочками вместо букв H и O, так как в файле **letters.txt** замена буквы H перманентна. При желании вы можете запустить программу 26 раз и заменить на * вообще все буквы.

Абсолютный минимум

Целью данной главы было объяснить принципы работы режима произвольного доступа к файлам. Если вы открываете файл в режиме произвольного доступа, вы можете считывать и записывать информацию в любой необходимой последовательности. Функция `fseek()` является встроенной функцией языка C, перескакивающей от позиции к позиции в файле.

Возможность изменять содержимое файла очень важна при необходимости обновления данных. Зачастую вам может понадобиться изменить адрес человека, обновить количество экземпляров товара, оставшихся на складе, и так далее, при этом вам не нужно перезаписывать весь файл, что было бы необходимо при использовании последовательного доступа. Далее приведены основные концепции этой главы.

- Если вам необходимо изменить данные файлы, воспользуйтесь знаком плюс при указании модификатора режима доступа к файлу функции `fopen()`.
- Запомните, что функция `fseek()` перемещает указатель по всему файлу произвольного доступа таким образом, что вы можете производить чтение или запись информации с начала, середины или конца файла.
- Не забудьте закрыть файл по завершении работы с ним.
- Не пытайтесь работать с файлом, если вызов функции `fopen()` завершился неудачей (функция вернула 0).

Глава 30
ОРГАНИЗАЦИЯ ПРОГРАММ С ПОМОЩЬЮ ФУНКЦИЙ

В этой главе

- Добавление функций в программу
- Выбор между глобальными и локальными переменными

Как правило, компьютерные программы — это нечто большее, чем 20–30 строк программного кода, которые вы встречаете в учебниках. В «реальном мире» компьютерные программы гораздо длиннее. Однако длинные программы содержат очень много кода, который только мешается во время обучения, именно поэтому до настоящего момента в этой книге вы встречали только довольно короткие программы, код которых целиком помещался внутри функции `main()`.

Если бы вы уместили весь код длинной программы внутрь функции `main()`, то в последующем при возникновении необходимости внесения изменений, найти нужный участок кода было бы довольно проблематично. Эта глава первая из трех, в которых мы изучим способы разделения ваших программ на части с использованием нескольких функций. Категоризация программного кода путем разбиения его на разделы упрощает не только написание, но также и техническую поддержку программ.

Люди вынуждены писать, изменять и исправлять код. Чем более удобочитаемым вы сделаете код создаваемой вами программы, разделив его на несколько функций, тем быстрее вы сможете вернуться с работы (где работаете программистом) домой и отдохнуть! Как вы увидите в дальнейшем, разделение программы на функции позволяет вам сосредоточиться именно на том участке кода, который требуется изменить.

С функциями языка С приходит форма

Язык С был спроектирован таким образом, чтобы спровоцировать у вас через использование функций модульное мышление. Программа на языке С не является простой монолитной длинной программой. Программа на языке С состоит из множества процедур, которые, как

вы знаете, называются *функциями*. Одна из функций программы (обязательная для любой программы и, как правило, реализуемая первой) называется `main()`.

Если ваша программа делает множество операций, разбейте ее на несколько функций. Каждая функция должна выполнять одну основную задачу. Например, если вы пишете программу на языке C, которая присваивала бы новому сотруднику идентификационный номер, принимала бы контактную информацию этого человека, а затем добавляла бы его или ее в платежную ведомость, вы *могли бы* написать весь этот код в одной большой функции — функции `main()`, — как показано в следующем черновике программного кода:

```
main()
{
   //Это не рабочая программа, просто черновик…
…
   //Первый участок кода, присваивающий новому
   //сотруднику ID
…
   //Следующий участок просит ввести контактные данные
…
   //Последний участок добавляет сотрудника
   //в платежную ведомость
…
   return (0);
}
```

Эта программа *не предлагает* хорошего варианта решения поставленных задач, поскольку она слишком последовательна. Несмотря на то, что требуется выполнить несколько ясно различимых задач, весь код записан внутри функции `main()`. Возможно, этой программе не понадобится большое количество строк кода, но всегда лучше завести себе привычку разбивать каждую программу на четко различимые задачи.

ПРИМЕЧАНИЕ

Разбиение программы на более мелкие функции называется *структурным программированием.*

Не нужно использовать функцию `main()` для решения абсолютно всех поставленных задач. На самом деле функцию `main()` лучше всего практически не использовать ни для чего другого, кроме вызова других функций. Лучшим способом организации этой программы было бы написать несколько отдельных функций, по одной для каждой из выполняемых программой задач. Конечно, не каждая функция должна состоять только из одной строки кода, но убедитесь в том, что каждая функция вашей программы — это отдельный строительный блок, и она выполняет только одну задачу.

Далее приведен более удачный вариант черновика только что описанной программы:

```
main()
{
  assignID(); //Присваивает уникальный ID новому
сотруднику
  buildContact(); //Ввод контактной информации
сотрудника
  payrollAdd(); //Добавить сотрудника в платежную
ведомость
  return 0;
}
/*Вторая функция, присваивающая уникальный ID новому
сотруднику*/
assingnID()
{
  //Блок кода на языке C, присваивающий новому
сотруднику
  //уникальный идентификационный номер
  return 0;
}
/*Следующая функция: построение контакта*/
buildContact()
{
  //Блок кода для ввода домашнего адреса, номера
  // телефона даты рождения и прочих данных сотрудника
  return 0;
```

```
}
/*Четвертая функция добавляет сотрудника в платежную
ведомость*/
payrollAdd()
{
  //Код для установки зарплаты сотрудника, его
  //премий и прочей информации в системе расчета
  //заработной платы
  return 0;
}
```

ПРИМЕЧАНИЕ

Несмотря на то, что этот черновик программного кода длиннее предыдущего, он лучше организован, а это значит, что его проще поддерживать. Единственное, что выполняет функция `main()` — контроль вызова других функций через демонстрацию их названий.

Каждая из функций выполняет свою задачу, а затем возвращает управление функции `main()`, которая вызывает следующую функцию, и так происходит до тех пор, пока не закончится список вызываемых функций, после чего функция `main()` возвращает управление вашей операционной системе. В программе функция `main()` выполняет роль, сходную с перечнем глав в начале книги. При наличии адекватных комментариев функция `main()` позволяет вам узнать, код какой именно функции необходимо изменить.

СОВЕТ

Хорошим правилом для вас было бы писать функции, содержащие количество строк кода, не превышающее максимальное количество, вмещающееся на экране. Если ваша функция длиннее, то, вероятно, вы перегружаете ее заданиями. Вспомните, разве в старшей школе вы не *ненавидели* читать книги с д-л-и-н-н-ы-м-и главами? В таком случае, вам не понравится работать с программами, содержащими длинные функции.

Любая функция может вызвать любую функцию. Например, если бы вы захотели, чтобы функция `buildContact()` распечатала полные контактные данные после завершения их ввода, вы могли бы за-

ставить функцию `buildContact()` вызвать другую функцию, например, с именем `printContact()`, по завершение работы функция `printContact()` вернула бы управление функции `buildContact()`. Далее приведен пример черновика такого кода.

```
main()
{
  assignID(); //Присваивает уникальный ID новому
сотруднику
  buildContact(); //Ввод контактной информации
сотрудника
  payrollAdd(); //Добавить сотрудника в платежную
ведомость
  return 0;
}
/*Вторая функция, присваивающая уникальный ID новому
сотруднику*/
assingnID()
{
  //Блок кода на языке C, присваивающий новому
сотруднику
  //уникальный идентификационный номер
  return 0;
}
/*Следующая функция: построение контакта*/
buildContact()
{
  //Блок кода для ввода домашнего адреса, номера
телефона
  //даты рождения и прочих данных сотрудника
  printContact();
  return 0;
}
/*Четвертая функция добавляет сотрудника в платежную
ведомость*/
payrollAdd()
```

```
{
  //Код для установки зарплаты сотрудника, его
  //премий и прочей информации в системе расчета
  //заработной платы
  return 0;
}
/*Пятая функция выводит полную контактную информацию
на экран*/
printContact()
{
  //Код для печати контактной информации
  return; //Возврат в buildContact(), а не в main()
}
```

ПРИМЕЧАНИЕ

Посмотрите на функции игры «Покер с обменом» из приложения Б. Код этой программы занимает лишь несколько страниц, но содержит сразу несколько функций. Просмотрите код программы и попытайтесь найти хотя бы одну функцию, вызывающую другую функцию, расположенную где-либо еще в коде программы.

Вся электронная промышленность научилась кое-чему у мира программирования. Сегодня большинство электронных устройств, таких как телевизоры, компьютеры и телефоны, содержат несколько плат, которые могут быть заменены, обновлены или удалены вовсе без затрагивания остальной системы. Аналогичным образом вы сможете изменить и некоторые компоненты вашей программы: если вы пишете хорошо структурированные программы, используя функции, вы сможете изменять только те функции, которые нуждаются в редактировании, но при этом не затрагивать не связанный с данными функциями программный код.

Локальная или глобальная?

Для работы черновику программы, приведенному в предыдущем разделе, требуется чуть больше кода. Прежде чем вы научитесь добавлять код, вам стоит более подробно взглянуть на объявление переменных.

В языке С все переменные могут быть либо *локальными*, либо *глобальными*. Все переменные, которые вы встречали до настоящего момента, были локальными. В большинстве случаев локальная переменная безопаснее глобальной, так как локальная переменная предоставляет доступ к себе по принципу «только тем, кто знает». Иными словами, если функции требуется переменная, то данная функция может получить доступ к локальным переменным другой функции благодаря механизму передачи переменных, описанному в следующей главе.

Если вам не нужно, чтобы одна функция получала доступ к переменным другой функции, она его и не получит. В то же время любая функция может считывать, изменять и обнулять значения глобальных переменных, поэтому глобальные переменные считаются менее безопасными.

Следующие правила описывают различия между локальными и глобальными переменными:

- Переменная считается *глобальной* только если эта переменная объявляется (например, `int i;`) перед именем функции.
- Переменная считается *локальной* только если вы определяете ее после открывающей фигурной скобки. У некоторых выражений, например, `while`, также есть фигурные скобки, и вы можете объявить локальные переменные внутри этих скобок.

СОВЕТ

Код, заключенный между открывающей и закрывающей фигурными скобками, называется *блоком*.

Принимая во внимание вышеприведенные правила, должно казаться очевидным, что в следующей программе переменные l1 и l2 являются локальными, а g1 и g2 — глобальными:

```
// Программа-пример №1 из главы 30
// Руководства по C для новичков, 3-е издание
// Файл Chapter30ex1.c
/* Эта программа является простой демонстрацией
различий между глобальными и локальными переменными.
*/
#include <stdio.h>
```

```
int g1 = 10;
main()
{
    float l1;
    l1 = 9.0;
    printf("%d %.2f\n", g1, l1); // выводит 1-ю
                // глобальную и локальную переменную
    prAgain(); // вызывает первую функцию
    return 0;
}
float g2 = 19.0;
prAgain()
{
    int l2 = 5;
    // Невозможно распечатать l1- это локальная
переменная ф-ции main
    printf("%d %.2f %d\n", l2, g2, g1);
    return;
}
```

СОВЕТ

Возможно, вы еще не разобрались в предназначении выражения `return 0;`, а в этом примере выражение return употребляется само по себе в конце тела функции `prAgain()`. В следующих двух главах вы найдете детальное описание выражения return.

Переменная `g2` является глобальной, так как она объявлена перед функцией `prAgain()`.

Локальные переменные могут использоваться *только* внутри того блока кода, в котором они были объявлены. Именно поэтому переменная `l1` не может быть ни изменена, ни распечатана функцией `prAgain()`, переменная `l1` является локальной переменной функции `main()`. По аналогии, переменную l2 нельзя использовать в функции `main()`, так как данную переменную может «видеть» только функция `prAgain()`. Переменная `g1` видна по всей программе, а переменная `g2` видна только коду, следующему *после* ее объявления.

СОВЕТ

Глобальные переменные видны по всему программному коду, следующему *после* их объявления. Не объявляйте глобальные переменные в середине программы (как это было сделано в предшествующем примере), так как объявление таких переменных может быть довольно сложно отыскать при отладке программы. Ограничьте использование глобальных переменных или вовсе откажитесь от них. Если же вы используете глобальные переменные, объявляйте их все перед функцией main(), где вам их будет очень легко найти (например, в ситуациях, когда вам нужно изменить их или уточнить тип данных).

В ранее приведенном черновике программы есть одна проблема: если вы используете только локальные переменные (а вы должны стараться поступать именно так), то идентификационный номер, созданный в функции `assignID()`, не может быть использован в функциях `buildContact()` и `addPayroll()`. Оставайтесь с нами — и в следующей главе мы покажем вам решение этой проблемы.

ВНИМАНИЕ

Если при компиляции предыдущей программы вы получите предупреждение о вызове функции без прототипа, не обращайте на него внимание: глава 32 ответит на все ваши вопросы.

Абсолютный минимум

Целью данной главы было обучить вас блочному подходу к написанию программ на языке C. Длинные программы могут стать громоздкими, если не разбивать их на несколько отдельных функций. Одна-единственная длинная функция `main()` подобна толстой книге без разделения на главы. Разбейте свои длинные программы на несколько отдельных функций так, чтобы каждая функция выполняла свою конкретную задачу.

При разбиении программы на отдельные функции вы должны задумываться над тем, каким образом переменные используются по всему программному коду. Локальные переменные объявляются внутри функции и могут быть использованы только внутри этой функции. Противоположностью локальной переменной является переменная глобальная,

значение которой может быть использовано всеми функциями после объявления этой переменной. Многие не одобряют использование глобальных переменных. Локальные переменные безопаснее глобальных, так как вы можете ограничить количество функций, могущих получить доступ к таким переменным. В следующей главе вы узнаете, каким образом функции могут совместно использовать локальные переменные. Далее приведены основные концепции этой главы.

- Объявляйте локальные переменные после открывающей фигурной скобки блока кода. Объявляйте глобальные переменные перед началом тела функции.
- Локальные переменные безопаснее глобальных, поэтому старайтесь как можно чаще использовать именно локальные переменные.
- Для упрощения поддержки и ускорения разработки разбивайте свои программы на несколько отдельных функций.
- Не объявляйте глобальные переменные посередине программы, так как такие переменные очень сложно отыскать впоследствии.
- Не начинайте написание программы с глобальных переменных. По мере роста вашей программы вы можете столкнуться с необходимостью добавить глобальную переменную — добавьте. (В следующей главе мы более подробно обсудим использование локальных переменных вместо глобальных.)

Глава 31
ПЕРЕДАЧА ПЕРЕМЕННЫХ В ФУНКЦИИ

В этой главе

- Передача аргументов в функцию
- Различие между передачей по значению и передачей по адресу

Предыдущая глава оставила открытыми несколько вопросов. Если использование множества функций — это хорошо (а это хорошо), и локальные переменные — это тоже хорошо (и это хорошо), значит, должен быть какой-то способ совместного использования локальных переменных несколькими функциями при возникновении такой необходимости (и такой способ есть).

Вам не нужно, чтобы *все* функции имели доступ ко *всем* переменным, так как не всем функциям нужен доступ ко всем переменным. Если необходимо, чтобы все функции имели полный доступ ко всем переменным, то вы также можете воспользоваться глобальными переменными.

Для совместного использования данных разными функциями, вы должны *передавать* переменные от функции к функции. Когда одна переменная передает переменную другой функции, только эти две функции имеют доступ к этой переменной (в том случае если переменная локальная). В этой главе объясняется, как осуществлять передачу переменных между функциями.

Передача аргументов

При передаче переменной из одной функции в другую вы осуществляете *передачу аргумента* из первой функции в следующую. Одновременно можно передавать сразу несколько переменных. Принимающая функция *принимает параметры* от функции, передающей переменные.

ВНИМАНИЕ

При передаче и получении значений слова *переменная*, *аргумент* и *параметр* могут использоваться как равнозначные синонимы. Имя гораздо менее важно понимания того, что происходит. Рисунок 31.1 помогает объяснить значение вышеприведенных терминов.

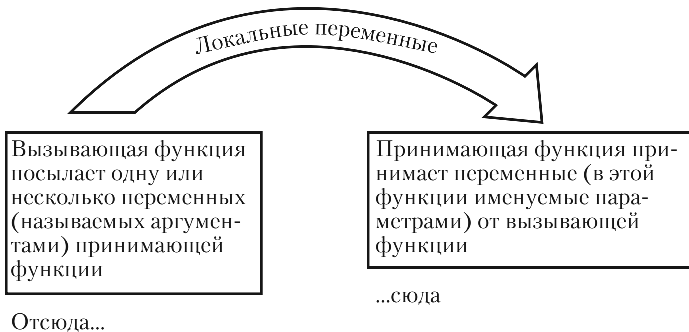


Рис. 31.1. Правильное понимание терминов

Методы передачи аргументов

Вы можете осуществить передачу аргументов от одной функции другой двумя способами: *по значению* и *по адресу*.

Оба метода передают аргументы *в* принимающую функцию от вызывающей функции. Кроме того, существует способ *вернуть* значение из функции обратно в вызывающую функцию (см. следующую главу).

Весь этот разговор о передаче значений сосредоточен на содержимом скобок, следующих после имени функции. Все верно, у этих пустых скобок, в конце концов, тоже есть свое предназначение!

Переменные, которые вы хотели бы передать, помещаются внутрь скобок операции вызова функции, а также в скобки принимающей функции, как вы увидите в следующем разделе.

ПРИМЕЧАНИЕ

Да, передача значений очень важна! Однако это все очень легко, как вы вскоре сами сможете убедиться.

Передача по значению

Иногда операцию *передача по значению* также называют *передача по копии*. Часто вы будете слышать употребление обоих терминов в одинаковом контексте, так как оба термина обозначают одно и тоже. Передача по значению означает, что *значение* переменной передается принимающей функции, но не сама переменная. Далее приведена программа, передающая значение из функции `main()` в функцию `half()`:

```
// Программа-пример №1 из главы 31
// Руководства по C для новичков, 3-е издание
// Файл Chapter31ex1.c
/* Эта программа демонстрирует передачу переменной
в функцию по значению */
#include <stdio.h>
main()
{
    int i;
    printf("Пожалуйста введите целое число... ");
    scanf(" %d", &i);
  // Вызов функции half с передачей значения переменной i
    half(i);
    // Покажем, что ф-ция не изменила значение i
    printf("В main(), i все еще равно %d.\n", i);
    return(0); // Конец программы
}
/*****************************************************/
/* Иногда разделители, как тот, что выше, — это
хороший способ четко визуально разбить код на
несколько функций. */
half (int i) // Принимает значение i
{
    i = i / 2;
    printf("Значение, поделенное пополам: %d.\n", i);
    return; // Возврат в main
}
```

Приведем результат тестового запуска программы:

```
Пожалуйста введите целое число… 28
Значение, поделенное пополам: 14.
В main(), i все еще равно 28.
```

Рассмотрим первую строку функции `half()`:

```
half (int i) // Принимает значение i
```

Обратите внимание, что вы должны включать тип данных (`int`) при указании списка параметров принимающие функции. На рис. 31.2 показано, что содержимое переменной `i` передается функции `half()`. Значение переменной `i` в функции `main()` остается неизменным, *так как другой функции передается только копия значения.*

```
main(
{
  int i;
  /* Код запроса и ввода данных */
   half(i);    значение i
   return 0;
}
                 half(int i)
                 {
                    i = i / 2
                    printf(/* Остальной код printf() */
                    return;
                 }
```

Рис. 31.2. Передается значение `i`, но не сама переменная `i`

ВНИМАНИЕ

Передача по значению защищает переменную. Если принимающая функция изменяет переменную, передаваемую по значению, значение переменной в вызывающей функции остается без изменений. Таким образом, передаваемое значение всегда в безопасности, так как принимающая функция не может изменять значения переменных вызывающей функции. Принимающая функция может только использовать эти значения.

Если бы в предыдущей программе параметр принимающей функции имел имя `i2`, программа работала бы точно также, как работает сейчас. Переменная `i2` была бы локальной для функции `half()`, тогда как пе-

ременная `i` из функции `main()` была бы локальной переменной функции `main()`. Переменная `i2` была бы локальной для функции `half()`, отличной от функции `main()`.

В языке С все переменные — не массивы передаются по значению. Таким образом, вы передаете функции переменную, не являющуюся массивом, то функции передаются только копия этой переменной. В вызывающей функции значение переменной не изменяется, вне зависимости от того, что делает с полученным значением вызванная функция.

Передача по адресу

Когда вы передаете массив в другую функцию, этот массив передается по адресу. Вместо копирования и передачи копии массива принимающей функции передается только адрес массива. После этого принимающая функция записывает адрес переданного массива *в соответствующий параметр*, таким образом, и принимающая и вызывающая функции работают с одним и тем же адресом массива. Если принимающая функция изменяет одну из переменных в списке параметров, *аргументы вызывающей функции также изменяются*.

Следующая программа передает массив в функцию. Принимающая функция записывает символ `X` во все элементы полученного массива, а функция `main()` выводит массив на печать. Обратите внимание, что функция `main()` выведет на экран строку из символов `X`, так как принимающая функция изменила значение аргумента:

```
// Программа-пример №2 из главы 31
// Руководства по C для новичков, 3-е издание
// Файл Chapter31ex2.c
/* Эта программа демонстрирует передачу переменной
в функцию по значению */
#include <stdio.h>
main()
{
    char name[15] = "Michael Hatton";
    change(name);
    printf("По возвращении в main(), имя изменилось
на: %s.\n", name);
    return(0); // Конец программы
```

```
}
/****************************************************/
/* Иногда разделители, как тот, что выше — это хороший
способ четко визуально разбить код на несколько функций. */
change(char name[15]) // Принимает значение i
{
    // Изменить строку по адресу, указанному name
    strcpy(name, "XXXXXXXXXXXXXX");
    return; // Возврат в main
}
```

На экране результат работы программы будет выглядеть следующим образом:

```
По возвращении в main(), имя изменилось на:
XXXXXXXXXXXXXX
```

Если вы хотите переопределить передачу переменных — не массивов по значению, вы можете заставить компьютер передать обычную переменную-не массив по адресу. Впрочем, такая операция выглядит весьма и весьма странно! Далее приведена программа, похожая на ту, что вы видели в начале главы, однако экранный вывод, производимый этой программой, отличается от вывода предшественницы:

```
// Программа-пример №3 из главы 31
// Руководства по C для новичков, 3-е издание
// Файл Chapter31ex3.c
/* Эта программа демонстрирует передачу переменной
в функцию по адресу */
#include <stdio.h>
main()
{
    int i;
    printf("Пожалуйста введите целое число... ");
    scanf(" %d", &i);
   // Вызов функции half с передачей адреса переменной i
    half(&i);
    // Покажем, что ф-ция изменила значение i
```

```
    printf("В main(), i все еще равно %d.\n", i);
    return(0); // Конец программы
}
/*****************************************************/
/* Иногда разделители, как тот, что выше — это хороший
способ четко визуально разбить код на несколько функций. */
half (int *i) // Принимает адрес i
{
    *i = *i / 2;
    printf("Значение, поделенное пополам: %d.\n", *i);
    return; // Возврат в main
}
```

Приведем текст, выводимый программой на экран:

```
Пожалуйста введите целое число… 28
Значение, поделенное пополам: 14.
В main(), i все еще равно 14.
```

Выглядит весьма необычно, но если вы хотите передать не массив по адресу, сопроводите его приставкой-оператором `&` (взятие по адресу) в передающей функции, а также вставляйте символ `*` (операция разыменования) перед именем переменной при *каждом употреблении* этой переменной в теле принимающей функции. Если вы думаете, что таким образом вы передаете в функцию указатель, то вы абсолютно правы.

ПРИМЕЧАНИЕ

Теперь функция `scanf()` уже не покажется столь непонятной. Помните, что при передаче переменных — не массивов в функцию `scanf()` вы должны использовать оператор `&` перед именем переменной, однако при передаче массива это делать не нужно. При вызове функции `scanf()` вы должны передать ей адреса переменных с тем, чтобы функция смогла их изменить. Так как строки являются массивами, при наборе строки с клавиатуры перед именем массива употреблять оператор взятия по адресу не нужно.

Следующая программа передает целочисленную переменную `i` по значению, переменную `x` типа `float` по значению и целочисленный массив также по значению (как и должны передаваться массивы):

```
// Программа-пример №4 из главы 31
// Руководства по C для новичков, 3-е издание
// Файл Chapter31ex4.c
/* Эта программа демонстрирует передачу в функцию
нескольких переменных */
#include <stdio.h>
// Объяснение следующего выражения см. в главе 32
changeSome(int i, float *newX, int iAry[5]);
main()
{
    int i = 10;
    int ctr;
    float x = 20.5;
    int iAry[] = {10, 20, 30, 40, 50};
    puts("Переменные main() перед вызовом функции:");
    printf("i равна %d\n", i);
    printf("x равна %.1f\n", x);
    for (ctr = 0; ctr < 5; ctr++)
    {
        printf("iAry[%d] равен %d\n", ctr, iAry[ctr]);
    }
    // Вызовем функцию changeSome, передав ей значение
i
    // и адрес x (употребив &)
    changeSome(i, &x, iAry);
   puts("\n\nПеременные main() после вызова функции:");
    printf("i равна%d\n", i);
    printf("x равна %.1f\n", x);
    for (ctr = 0; ctr < 5; ctr++)
    {
        printf("iAry[%d] равен %d\n", ctr, iAry[ctr]);
    }
    return(0); // Конец программы
}
```

```
/*****************************************************/
changeSome (int i, float *newX, int iAry[5])
{
    // Все переменные изменяются, но только float
    // и массив сохраняют изменения, когда программа
    // возвращается в main()
    int j;
    i = 47;
   *newX = 97.6; // Та же ячейка памяти, что и у x в main
    for (j = 0; j < 5; j++)
    {
        iAry[j] = 100 + 100*j;
    }
    return; // Возврат в main
}
```

Далее приведен текст, выводимый программой на экран компьютера:

```
Переменные main() перед вызовом функции:
i равна 10
x равна 20.5
iAry[0] равен 10
iAry[1] равен 20
iAry[2] равен 30
iAry[3] равен 40
iAry[4] равен 50
Переменные main() после вызова функции:
i равна 10
x равна 97.6
iAry[0] равен 100
iAry[1] равен 200
iAry[2] равен 300
iAry[3] равен 400
iAry[4] равен 500
```

Следующая глава завершит рассказ о передаче значений между функциями, показав вам способы возврата значений из одной функции в другую. Кроме того, вы наконец поймете истинное предназначение файла `stdio.h`.

Абсолютный минимум

Целью этой главы было показать вам, каким образом функции могут совместно использовать локальные переменные. Когда одной функции требуется доступ к локальной переменной, определенной в другой функции, вы должны передать ей эту переменную. В скобках, следующих сразу за именем функции, приводится перечень переменных, которые ваша функция передает или принимает.

Как правило, переменные — не массивы передаются *по значению*, а это значит, что принимающая функция может использовать эти переменные, но не может изменить их значения в вызывающей функции. Массивы передаются *по адресу*, это значит, если принимающая функция изменяет их, то значения изменяются также и в вызывающей функции. Переменные — не массивы также могут быть переданы по адресу, для этого именам передаваемых таким образом переменных должен предшествовать оператор взятия по адресу `&`, а в принимающей функции — оператор разыменования `*`. Далее приведены основные концепции текста главы.

- Если хотите, чтобы функции совместно использовали локальные переменные, организуйте их передачу из одной функции в другую.
- Если хотите защитить значения переменных от изменения принимающей функцией, передавайте переменные по значению.
- Если вы хотите, чтобы принимающая функция изменила значения передаваемых переменных, их следует передавать по адресу.
- Если вы хотите передать переменную — не массив по адресу, поставьте знак `&` перед ее именем. Не употребляйте оператор `&` при передаче массивов.
- Не передавайте переменную-массив по значению: в С нет механизмов для такой передачи.

Глава 32
ВОЗВРАТ ДАННЫХ ИЗ ФУНКЦИЙ

В этой главе

- Возвращение значений
- Использование типа данных `return`

Это глава — отнюдь не конец вашего обучения программированию на языке C, наоборот — это только начало! Глубокая фраза, не так ли? Эта глава вносит последние штрихи в нашу картину полифункционального программирования, показывая вам как возвращать значения из вызванных функций в вызывающую. Кроме того, в главе объясняется, что такое *прототипы* функций.

Итог вот в чем: сейчас вы поймете, почему большинство программ из этой книги содержали вот эту строку:

```
return 0;
```

Кроме того, вы поймете истинное предназначение заголовочных файлов.

Возврат значений

К настоящему моменту вы уже узнали, как отправлять переменные в функцию, теперь вы готовы изучить механизмы возврата значений. Если функция должна вернуть значение, воспользуйтесь выражением `return` для возврата этого значения. Программисты, работающие на языке C, зачастую используют круглые скобки после выражения `return`, заключая в эти скобки возвращаемое значение, например, `return(answer);`.

ПРИМЕЧАНИЕ

Если функция не возвращает никаких значений, то в выражении `return` нет необходимости, так как функция автоматически вернет управление вызывающей функции. Однако если вы хотите вернуть значение, то выражение `return` является обязательным.

Несмотря на то что вы можете передать в функцию сразу несколько значений, вернуть в вызывающую функцию можно *только одно значение*. Рисунок 32.1 объясняет механизм возврата значений. В этом правиле нет исключений.

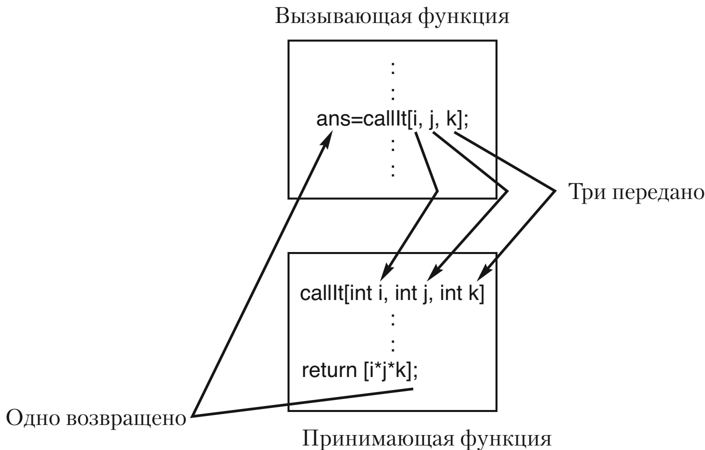


Рис. 32.1. Вы можете передать в функцию сразу несколько значений, но вернуть — лишь одно

Хотя единственное возвращаемое значение может показаться весьма ограничивающим, на самом деле это не так. Рассмотрим встроенную функцию `sqrt()`, рассмотренную нами в главе 20.

Функция `sqrt()` возвращает квадратный корень из любого передаваемого ей значения. Функция `sqrt()` возвращает не несколько значений, а всего одно. На самом деле ни одна из встроенных функций не возвращает несколько значений, ваши функции этого тоже не смогут.

ПРИМЕЧАНИЕ

Может показаться, что функция `gets()` возвращает несколько значений, потому что эта функция возвращает строку — массив символов. Однако не забывайте, что имя массива — это лишь указатель на его первый элемент, таким образом функция `gets()` возвращает символьный указатель на начало введенной пользователем строки символов.

Следующая программа содержит функцию, принимающую три числа с плавающей точкой: `test1`, `test2` и `test3`. Функция с именем

gradeAve() вычисляет средний балл из трех оценок и возвращает полученный ответ.

```
// Программа-пример №1 из главы 32
// Руководства по С для новичков, 3-е издание
// Файл Chapter32ex1.c
/* Эта программа демонстрирует возврат значений
функциями путем передачи трех чисел с плавающей
точкой (оценок) и вычисления среднего значения
из этих трех. */
#include <stdio.h>
float gradeAve(float test1, float test2, float test3);
main()
{
    float grade1, grade2, grade3;
    float average;
    printf("Какова Ваша оценка за первый тест? ");
    scanf(" %f", &grade1);
    printf("Какова Ваша оценка за второй тест? ");
    scanf(" %f", &grade2);
    printf("Какова Ваша оценка за третий тест? ");
    scanf(" %f", &grade3);
    //Передать в функцию три оценки и вернуть
    //средний балл
    average = gradeAve(grade1, grade2, grade3);
    printf("\nСредний балл за три теста составил:
            %.2f", average);
    return 0;
}
/**************************************************/
float gradeAve(float test1, float test2, float test3)
// Принимает значения трех оценок
{
    float localAverage;
    localAverage = (test1+test2+test3)/3;
```

```
    return (localAverage); // Возврат среднего
                           // балла в main()
}
```

Далее приведен результат тестового запуска программы:

```
Какова Ваша оценка за первый тест? 95
Какова Ваша оценка за второй тест? 88
Какова Ваша оценка за третий тест? 91
Средний балл за три теста составил: 91.33
```

ПРИМЕЧАНИЕ

Обратите внимание, что в функции `main()` функция `gradeAve()` присваивается переменной `average` и возвращает в нее полученное значение. В это время функция `main()` также должна выполнить несколько операций со значением, возвращенным функцией `gradeAve()`.

Кроме переменных после выражения return вы также можете поместить выражение. Например, следующий код:

```
sales = quantity * price;
return (sales);
```

идентичен нижеследующему:

```
return (quantity * price);
```

Тип данных *return*

В самом начале функции `gradeAve()` вы можете видеть ключевое слово `float`. `float` — это тип данных, возвращаемого функцией `gradeAve()` значения. Вы *должны* указывать тип возвращаемых данных перед каждой функцией, возвращающей значения. Если бы обсуждаемая функция возвращала значение типа `long int`, то перед именем функции следовало бы указать `long int`.

Если вы не указываете тип возвращаемого значения, компилятор автоматически предполагает тип `int`. Таким образом, компьютер будет ожидать, что каждая функция, тип возвращаемых данных для которой

не указан эксплицитно (явно), возвращает значение типа `int`. Для компилятора первые строки обеих функций равнозначны:

```
int myFun(int a, float x, char c)
```

и

```
myFun(int a, float x, char c) /* int предполагается */
```

СОВЕТ

Угадайте, что? Предполагается, что даже функция `main()` должна вернуть значение `int`, если вы не переопределили тип возвращаемого значения. *Именно поэтому* вы видели выражение `return 0;` в конце почти всех программ! Так как тип возвращаемого функцией `main()` значения не указан, предполагается возврат значения типа `int`, а выражение `return 0;` гарантирует, что в операционную систему будет возвращено именно значение типа `int`.

Если ваша функция не возвращает значений или если ей не передаются никакие значения, вы можете употребить ключевое слово `void` либо вместо возвращаемого типа данных, либо вместо списка параметров, либо и для того, и для другого. Таким образом, первая строка функции, которая не принимает никаких значений и ничего не возвращает, может выглядеть следующим образом:

```
void doSomething(void) /*Ничто не передается и не
возвращается*/
```

ВНИМАНИЕ

При использовании ANSI C функция `main()` не может иметь тип `void`, данная функция должна возвращать тип `int`. (Однако большинство компиляторов, даже те, что в инструкции к которым заявлено полное соответствие стандартам ANSI C, позволяют вам указывать `void` в качестве типа возвращаемого функцией `main()` значения.)

Последний шаг: прототип

Чтобы функция работала правильно нужно предпринять один последний шаг. Если функция возвращает значение любого типа, отличного от `int`, то вы должны создать прототип этой функции. На самом деле,

для ясности кода, вам следует создавать прототипы всех функций, в том числе тех, что возвращают целые числа.

Значение слова *прототип* — модель чего-то еще. Прототип функции — это всего лишь модель настоящей функции. Поначалу может показаться, что прототип C — это бесполезная трата времени.

Причина, по которой функции, возвращающие значение типа `int`, не нуждаются в прототипах, заключается в том, что `int` — это возвращаемое значение, прототипизированное по умолчанию возвращаемое значение, если не вами указано иное. Таким образом, следующие *два* прототипа служат для создания модели одной и той же функции:

```
int aFunc (int x, float y); /*2 передано, одно целое
возвращено*/
```

и

```
aFunc (int x, float y); /*2 передано, одно целое
возвращено*/
```

Если ваша функция возвращает целое число или вовсе не возвращает значений, прототипы не обязательны, но *настоятельно рекомендованы*. При создании прототипа, компилятор гарантирует, что вы не передадите функции значение типа `float`, если функция ожидает передачи значения типа. Без прототипа компилятор попытается конвертировать `float` в `char` — и в результате в функцию будет передано неверное значение.

Для создания прототипа функции поместите точную копию первой строки функции где-нибудь перед `main()`. В программе, которую вы видели ранее, прототип функции `gradeAve()` появляется прямо перед функцией `main()`. Данная строка *не является* вызовом функции, так как она находится перед функцией `main()`. Строка не является первой строкой настоящей функции из-за того, что она завершается точкой с запятой, как и все прототипы функций. Данная строка является прототипом функции. Если ваша программа вызывает 20 функций, то вы должны указать 20 прототипов.

Создавайте прототипы для всех функций в программе — для всех функций, вызываемых программой, даже для встроенных функций, таких как `printf()`. «А-а-а?» был бы резонным вопросом в данном случае. Вы можете задаться вопросом, как *вы* можете создать прототип для функцию, код для которой никогда не писали. Файл `stdio.h` содержит прототипы `printf()`, `scanf()`, `getchar()` и многих-многих других

функций ввода и вывода. Прототип функции `strcpy()` находится в файле `string.h`. При изучении новой встроенной функции всегда интересуйтесь, какой заголовочный файл ей соответствует, чтобы вы могли подключить нужный файл к программе с помощью директивы `#include` и быть уверенным, что все используемые в программе функции имеют прототип.

СОВЕТ

Так как функция `main()` — это первая функция программы, ей не нужен прототип. Функция `main()` известна как функция с автоматическим созданием прототипа, так как никакая другая функция не может вызывать функцию `main()` до момента появления последней в программном коде.

Следующая программа работает некорректно, так как типу данных `float` не сопоставлен корректный прототип. Запомните, что компилятор предполагает возвращение значения типа `int` (даже если ваша функция возвращает другой тип данных), если эта настройка не переопределена в прототипе функции.

```
#include <stdio.h>
compNet (float atomWeight, float factor);
main()
{
  float atomWeight, factor, netWeight;
  printf("Введите атомную массу: ");
  scanf(" %f", &atomWeight);
  printf("Введите атомный фактор: ");
  scanf(" %f", &factor);
  netWeight = compNet(atomWeight, factor);
  printf("Чистая масса составляет: %.4f\n",
netWeight);
  return 0;
}
/*****************************************************/
compNet(float atomWeight, float factor)
{
```

```
  float netWeight;
  netWeight = (atomWeight - factor) * .8;
  return (netWeight);
}
```

Далее приведен неправильный вывод программы:

```
Введите атомную массу: .0125
Введите атомный фактор: .98
Чистая масса составляет: 0.0000
```

Для решения проблемы измените прототип функции следующим образом:

```
float compNet (float atomWeight, float factor);
```

Также измените и первую строку функции (строка определения) с тем, чтобы она соответствовала прототипу:

```
float compNet (float atomWeight, float factor)
```

Подведем итоги

При наличии глобальных переменных в программе никогда не передавайте и не возвращайте их: глобальным переменным не требуется передача. Кроме того, список параметров вызывающей функции, принимающей функции, а также прототипа должны совпадать по количеству и типу данных. (Совпадение имен переменных необязательно.)

Теперь вы знаете все о передаче параметров и возвращении значений. Скорее надевайте свою думательную шапочку официального программиста и запускайте компилятор языка С!

Абсолютный минимум

Целью данной главы было отшлифовать ваши знания о функциях, рассказав про прототипы и возврат значений. Если ваша программа содержит несколько функций, создайте прототип для каждой из этих функций и поместите их перед функцией `main()`. Прототипу сообщают компилятору, что ему следует ожидать.

Прототип позволяет гарантировать, что вы по ошибке не передадите функции неправильный тип данных. Например, если прототипом установлено, что в функцию будет передано два значения типа `float`, но вы ошибочно пытаетесь передать функции два целых числа, компилятор выдаст предупреждение. Компилятор не будет выдавать предупреждений, если у функции нет прототипа, из-за чего вы можете получить неверные результаты.

Теперь, зная как возвращать значения, вы можете создавать функции, являющиеся копиями некоторых встроенных функций, например `sqrt()` или `rand()`. Вызываемая функция возвращает значение, полученное в результате выполнения программного кода этой функции. Функция, в том числе и встроенная, может вернуть не более одного значения. Ниже приведены ключевые концепции текста этой главы.

- Поместите тип возвращаемых данных перед именем функции, возвращающей данные.
- Возвращаемое значение следует после выражения return.
- Вызывающая функция должна производить какие-то действия над возвращенным значением, например, выводить его на экран или присваивать какой-то переменной. Вызов функции, возвращающей значение, без работы с этим значением бесполезен.
- Используйте ключевое слово `void` в качестве тип возвращаемого значения или в списке параметров функции, которая не возвращает или не принимает никаких значений.
- Не возвращайте из функции больше одного значения
- Не возвращайте нецелочисленный тип данных без прототипа. Лучше всего создавайте прототипы для *всех* функций, кроме `main()`.

Приложение A
ТАБЛИЦА ASCII

Dec	Hex	ASCII	Dec	Hex	ASCII
0	00	null	14	0E	♫
1	01	☺	15	0F	✲
2	02	☻	16	10	►
3	03	♥	17	11	◄
4	04	✦	18	12	↕
5	05	♣	19	13	‼
6	06	♠	20	14	¶
7	07	•	21	15	§
8	08	◘	22	16	–
9	09	◦	23	17	↨
10	0A	◙	24	18	↑
11	0B	♂	25	19	↓
12	0C	♀	26	1A	→
13	0D	♪	27	1B	←

Dec	Hex	ASCII	Dec	Hex	ASCII
28	1C	∟	59	3B	;
29	1D	↔	60	3C	<
30	1E	▲	61	3D	=
31	1F	▼	62	3E	>
32	20	пробел	63	3F	?
33	21	!	64	40	@
34	22	“	65	41	A
35	23	#	66	42	B
36	24	$	67	43	C
37	25	%	68	44	D
38	26	&	69	45	E
39	27	‘	70	46	F
40	28	(	71	47	G
41	29	)	72	48	H
42	2A	*	73	49	I
43	2B	+	74	4A	J
44	2C	‘	75	4B	K
45	2D	-	76	4C	L
46	2E	.	77	4D	M
47	2F	/	78	4E	N
48	30	0	79	4F	O
49	31	1	80	50	P
50	32	2	81	51	Q
51	33	3	82	52	R
52	34	4	83	53	S
53	35	5	84	54	T
54	36	6	85	55	U
55	37	7	86	56	V
56	38	8	87	57	W
57	39	9	88	58	X
58	3A	:	89	59	Y

Dec	Hex	ASCII	Dec	Hex	ASCII
90	5A	Z	121	79	y
91	5B	[	122	7A	z
92	5C	\	123	7B	{
93	5D	]	124	7C	¦
94	5E	^	125	7D	}
95	5F	–	126	7E	~
96	60	`	127	7F	Δ
97	61	a	128	80	Ç
98	62	b	129	81	ü
99	63	c	130	82	é
100	64	d	131	83	â
101	65	e	132	84	ä
102	66	f	133	85	à
103	67	g	134	86	å
104	68	h	135	87	ç
105	69	i	136	88	ê
106	6A	j	137	89	ë
107	6B	k	138	8A	è
108	6C	l	139	8B	ï
109	6D	m	140	8C	î
110	6E	n	141	8D	ì
111	6F	o	142	8E	Ä
112	70	p	143	8F	Å
113	71	q	144	90	É
114	72	r	145	91	æ
115	73	s	146	92	Æ
116	74	t	147	93	ô
117	75	u	148	94	ö
118	76	v	149	95	ò
119	77	w	150	96	û
120	78	x	151	97	ù

Dec	Hex	ASCII	Dec	Hex	ASCII
152	98	ÿ	181	B5	╡
153	99	Ö	182	B6	╢
154	9A	Ü	183	B7	╖
155	9B	¢	184	B8	╕
156	9C	£	185	B9	╣
157	9D	¥	186	BA	║
158	9E	₧	187	BB	╗
159	9F	ƒ	188	BC	╝
160	A0	á	189	BD	╜
161	A1	í	190	BE	╛
162	A2	ó	191	BF	┐
163	A3	ú	192	C0	└
164	A4	ñ	193	C1	┴
165	A5	—	194	C2	┬
166	A6	ª	195	C3	├
167	A7	º	196	C4	─
168	A8	¿	197	C5	+
169	A9	⌐	198	C6	╞
170	AA	¬	199	C7	╟
171	AB	½	200	C8	╚
172	AC	¼	201	C9	╔
173	AD	¡	202	CA	╩
174	AE	«	203	CB	╦
175	AF	»	204	CC	╠
176	B0		205	CD	═
177	B1		206	CE	╬
178	B2		207	CF	╧
179	B3	│	208	D0	╨
180	B4	┤	209	D1	╤

Dec	Hex	ASCII
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀
224	E0	α
225	E1	β
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	γ
232	E8	Φ
233	E9	θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	ø
238	EE	∈
239	EF	∩
240	F0	Å
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	•
250	FA	·
251	FB	√
252	FC	^{n}
253	FD	^{2}
254	FE	■
255	FF	

Приложение Б
ПРОГРАММА «ПОКЕР С ОБМЕНОМ»

Неправда, что программирование — это только работа, где совсем нет места игре, и следующая игра «Покер с обменом» тому доказательство! Программа-игра предоставляет длинный пример, который вы можете изучать по мере освоения программирования на языке С. Несмотря на то, что эта игра очень простая и прямолинейная, в программе происходит очень многое.

Как и в случае со всеми хорошо написанными программами, эта программа сопровождена подробными комментариями. На самом деле если вы прочли все главы данной книги, то вы поймете суть программного кода игры «Покер с обменом». Одной из причин сохранения простоты данной программы была попытка сохранить независимость этой программы от выбранного вами компилятора. Вы также можете попытаться выяснить, каким образом ваш компилятор изменяет цвета экранного шрифта, дабы сделать игру более веселой. Кроме того, когда вы изучите достаточное количество возможностей языка С, чтобы понимать внутреннюю работу программы, вы, возможно, захотите исследовать графические возможности и на самом деле нарисовать карты.

ПРИМЕЧАНИЕ

Вы также может поэкспериментировать и поизменять код программы. Например, некоторые покерные программы выдают вознаграждение за пару только в том случае, если имеющаяся у игрока пара — это два валета или выше (то есть два валета, две дамы, два короля или два туза). Каким образом вы поменяли бы функцию `analyzeHand()`, чтобы внести такое изменение?

```
// Программа-пример из приложения В
// Руководства по С для новичков, 3-е издание
// Файл AppendixBpoker.c
/* Программа представляет собой игру «Покер
с обменом». Пользователи могут играть так часто,
```

```
как захотят, делая ставки величиной от 1 до 5.
Пользователям выдается по 5 карт, после чего
пользователь должен решить, какие карты оставить,
а какие заменить. После этого происходит оценка вновь
выданных карт — и пользователь получает вознаграждение
в зависимости от ценности имеющихся  у него на руках
карт. Далее программа отображает на экране новый
банкролл пользователя, и ему дается возможность
продолжить игру */
// Заголовочные файлы
#include <stdio.h>
#include <time.h>
#include <ctype.h>
#include <stdlib.h>
// Объявление двух констант для определения, имеется
// ли у пользователя на руках флеш или стрит.
#define FALSE 0
#define TRUE 1
// Прототипы функций
void printGreeting();
int getBet();
char getSuit(int suit);
char getRank(int rank);
void getFirstHand(int cardRank[], int cardSuit[]);
void getFinalHand(int cardRank[], int cardSuit[], int
finalRank[],
                    int finalSuit[], int
ranksinHand[],
int suitsinHand[]);
int analyzeHand(int ranksinHand[], int suitsinHand[]);
main()
{
    int bet;
    int bank = 100;
    int i;
  int cardRank[5]; // Одно из 13 значений (Туз-Король)
  int cardSuit[5]; // Одно из 4 значений (для Пик,
                   // Бубен, Черв, Треф)
```

```
int finalRank[5];
int finalSuit[5];
int ranksinHand[13]; // Используется для оценки
                     // последней раздачи
int suitsinHand[4];  // Используется для оценки
                     // последней раздачи
int winnings;
time_t t;
char suit, rank, stillPlay;
// Функция вызывается вне цикла do...while,
// так как приветствие нужно отобразить
// только однажды, тогда как все остальное
// в функции main будет запущено
// несколько раз, в зависимости от того, сколько
// раз пользователь захочет сыграть.
printGreeting();
// Цикл запускается каждый раз, когда пользователь
// запрашивает раздачу новой партии карт
do {
    bet = getBet();
    srand(time(&t));
    getFirstHand(cardRank, cardSuit);
    printf("Ваши пять карт: \n");
    for (i = 0; i < 5; i++)
    {
     suit = getSuit(cardSuit[i]);
     rank = getRank(cardRank[i]);
     printf("Карта №%d: %c%c\n", i+1, rank, suit);
    }
    // Эти два массива используются, чтобы
    // провести оценку карт, выданных
     // пользователю. Однако они должны быть
    // обнулены, если пользователь затребует
    // несколько раздач.
    for (i=0; i < 4; i++)
    {
```

```
            suitsinHand[i] = 0;
        }
        for (i=0; i < 13; i++)
        {
            ranksinHand[i] = 0;
        }
        getFinalHand(cardRank, cardSuit, finalRank,
                 finalSuit, ranksinHand, suitsinHand);
        printf("Ваша последняя партия карт: \n");
        for (i = 0; i < 5; i++)
        {
         suit = getSuit(finalSuit[i]);
         rank = getRank(finalRank[i]);
         printf("Карта №%d: %c%c\n", i+1, rank, suit);
        }
      winnings = analyzeHand(ranksinHand,suitsinHand);
      printf("Вы выиграли %d!\n", bet*winnings);
      bank = bank - bet + (bet*winnings);
      printf("\nВ вашем банке сейчас %d.\n", bank);
      printf("\nХотите сыграть ещё раз? ");
      scanf(" %c", &stillPlay);
    } while (toupper(stillPlay) == 'Y');
    return;
}
/******************************************************/
// Распечатать быстрое приветствие, а также рассказать
// пользователям о ценности различных выигрышных
// комбинаций
void printGreeting()
{
    printf("************************************\n");
    printf("\n\n\tДобро пожаловать в казино для
новичков!\n\n");
    printf("\tДом видео покера с обменом\n\n");
    printf("************************************\n");
    printf("Правила таковы:\n");
```

```
    printf("Ваш начальный баланс 100 кредитов, вы
должны сделать ");
    printf("ставку в размере от 1 до 5 кредитов.\n");
    printf("Вам выдано 5 карт, вы должны выбрать,
какие ");
    printf("карты оставить, а какие - ");
    printf("сбросить.\n");
    printf("Вам нужно собрать наилучшую из возможных
комбинаций.\n");
    printf("\nДалее приведена таблица выигрышей
(в расчете на ");
    printf("ставку в 1 кредит):");
    printf("\nПара\t\t\t\t1 кредит");
    printf("\nДве пары\t\t\t2 кредита");
    printf("\nТри карты одного типа\t\t\t3 кредита ");
    printf("\nСтрит\t\t\t4 кредита");
    printf("\nФлэш\t\t\t\t5 кредитов");
    printf("\nФулхауз\t\t\t8 кредитов ");
    printf("\nЧетыре карты одного типа\t\t\t10
кредитов");
    printf("\nФлэш-рояль\t\t\t20 кредитов");
    printf("\n\nРазвлекайтесь!!\n\n");
}
// Функция для выдачи первых пяти карт
void getFirstHand(int cardRank[], int cardSuit[])
{
    int i,j;
    int cardDup;
    for (i=0; i < 5; i++)
    {
        cardDup = 0;
        do {
      // Один из 13 номиналов карты (2-10, В, Д, К, Т)
            cardRank[i] = (rand() % 13);
            // Одна из четырех мастей карты
            // (пики, бубны, червы, трефы)
            cardSuit[i] = (rand() % 4);
```

```
      // Цикл, гарантирующий уникальность каждой карты
            for (j=0; j < i; j++)
            {
                if ((cardRank[i] == cardRank[j]) &&
                    (cardSuit[i] == cardSuit[j]))
                {
                    cardDup = 1;
                }
            }
        } while (cardDup == 1);
    }
}
// Функция, заменяющая целочисленный номер масти на
// букву, представляющую эту масть
char getSuit(int suit)
{
    switch (suit)
    {
        case 0:
            return('п');
        case 1:
            return('б');
        case 2:
            return('ч');
        case 3:
            return('т');
    }
}
// Функция, заменяющая целочисленный номер номинала
// карты на букву, представляющую этот номинал
char getRank(int rank)
{
    switch (rank)
    {
        case 0:
            return('T');
```

```
        case 1:
            return('2');
        case 2:
            return('3');
        case 3:
            return('4');
        case 4:
            return('5');
        case 5:
            return('6');
        case 6:
            return('7');
        case 7:
            return('8');
        case 8:
            return('9');
        case 9:
            return('10');
        case 10:
            return('B');
        case 11:
            return('Д');
        case 12:
            return('K');
    }
}
// Функция для получения ставки пользователя (от 1 до 5)
int getBet()
{
    int bet;
    do //Будет повторяться, пока пользователь не
введет 0-5
    {
        printf("Сколько вы хотите поставить? (Введите
число от ");
        printf("1 до 5 или 0 для выхода из игры): ");
```

```
        scanf(" %d", &bet);
        if (bet >= 1 && bet <= 5)
        {
            return(bet);
        }
        else if (bet == 0)
        {
            exit(1);
        }
        else
        {
            printf("\n\nПожалуйста введите ставку от 1
до 5 или ");
            printf("0 для выхода из игры.\n");
        }
    } while ((bet < 0) || (bet > 5));
}
// Последняя функция проводит оценку последней
// выданной игроку партии карт
int analyzeHand(int ranksinHand[], int suitsinHand[])
{
    int num_consec = 0;
    int i, rank, suit;
    int straight = FALSE;
    int flush = FALSE;
    int four = FALSE;
    int three = FALSE;
    int pairs = 0;
    for (suit = 0; suit < 4; suit++)
        if (suitsinHand[suit] == 5)
            flush = TRUE;
        rank = 0;
        while (ranksinHand[rank] == 0)
            rank++;
        for (; rank < 13 && ranksinHand[rank]; rank++)
            num_consec++;
```

```
        if (num_consec == 5) {
            straight = TRUE;
        }
        for (rank = 0; rank < 13; rank++) {
            if (ranksinHand[rank] == 4)
                four = TRUE;
            if (ranksinHand[rank] == 3)
                three = TRUE;
            if (ranksinHand[rank] == 2)
                pairs++;
        }
        if (straight && flush) {
            printf("Флеш-рояль\n\n");
            return (20);
        }
        else if (four) {
            printf("Четыре одного типа\n\n");
            return (10);
        }
        else if (three && pairs == 1) {
            printf("Фуллхаус\n\n");
            return (8);
        }
        else if (flush) {
            printf("Флеш\n\n");
            return (5);
        }
        else if (straight) {
            printf("Стрит\n\n");
            return (4);
        }
        else if (three) {
            printf("Три одного типа\n\n");
            return (3);
        }
        else if (pairs == 2) {
```

```
            printf("Две пары\n\n");
            return (2);
        }
        else if (pairs == 1) {
            printf("Пара\n\n");
            return (1);
        }
        else {
            printf("Старшая карта\n\n");
            return (0);
        }
}
// Функция просматривает все пять карт первой партии
// и спрашивает пользователя, хочет ли он оставить
// карту. Если нет, то карта пользователя заменяется.
void getFinalHand(int cardRank[], int cardSuit[], int
finalRank[],
                   int finalSuit[], int ranksinHand[],
                    int suitsinHand[])
{
    int i, j, cardDup;
    char suit, rank, ans;
    for (i=0; i < 5; i++)
    {
        suit = getSuit(cardSuit[i]);
        rank = getRank(cardRank[i]);
        printf("Хотите сохранить карту №%d: %c%c?",
i+1, rank, suit);
        printf("\nПожалуйста, ответьте (Y/N): ");
        scanf(" %c", &ans);
        if (toupper(ans) == 'Y')
        {
            finalRank[i] = cardRank[i];
            finalSuit[i] = cardSuit[i];
            ranksinHand[finalRank[i]]++;
            suitsinHand[finalSuit[i]]++;
```

```
            continue;
        }
        else if (toupper(ans) == 'N')
        {
            cardDup = 0;
            do {
                cardDup = 0;
                finalRank[i] = (rand() % 13);
                finalSuit[i] = (rand() % 4);
            // Сравнить новую карту с 5 картами первой
            // партии для во избежание повторов
                for (j=0; j < 5; j++)
                {
                  if ((finalRank[i] == cardRank[j]) &&
                        (finalSuit[i] == cardSuit[j]))
                    {
                        cardDup = 1;
                    }
                }
                // Далее, сопоставить новую карту
                // с только что
                // выданными во избежание повторов
                for (j=0; j < i; j++)
                {
                 if ((finalRank[i] == finalRank[j]) &&
                       (finalSuit[i] == finalSuit[j]))
                    {
                        cardDup = 1;
                    }
                }
            } while (cardDup == 1);
            ranksinHand[finalRank[i]]++;
            suitsinHand[finalSuit[i]]++;
        }
    }
}
```

Об авторах

Грег Перри — лектор и автор печатных изданий как по программированию (то есть созданию), так и применению компьютерных приложений. Он известен упрощением сложных тем программирования до уровня начинающих программистов. Перри занимается обучением студентов и практическим программированием на протяжении двух десятилетий. Он получил первую научную степень по информатике, после чего получил степень магистра в области корпоративного финансирования. Однако кроме написания публикаций он проводит консультации и посещает с лекциями различные уголки Соединенных Штатов, в том числе принимает участие в престижных и известных конференциях по программированию, посвященных разработке программного обеспечения (Software Development). Перри является автором более 75 различных книг по компьютерным тематикам. В свободное время он также проводит семинары, посвященные туризму и путешествиям по Италии, его второму любимому месту на Земле.

Дин Миллер — писатель и редактор с более чем 20-летним стажем работы как в издательском деле, так и в области лицензированной потребительской продукции. На протяжении этих лет он создал сам или помог другим придать форму книжным бестселлерам и сериям книг, в том числе «Teach Yourself in 21 Days», «Teach Yourself in 24 Hours», а также серия книг «Unleashed», которые были впервые выпущены издательством «Sams Publishing». Он является автором книг по программированию на языке С, а также профессиональному реслингу, и он все еще находится в поисках способа сочетания обоих направлений в очень своеобразный сплав.

Благодарности

Грег: «Я хотел бы поблагодарить всех своих друзей из Пирсон. Большинство писателей сослались бы на них, назвав их редакторами, для меня же они просто друзья. Я хотел, чтобы все мои читатели понимали, что работники Пирсон больше всех заботятся о вас. То, что они делают, является результатом их учета ваших знаний и вашего удовольствия.

Лично хотелось бы поблагодарить мою прекрасную невесту Джейн, мою мать Бетти Перри и моих друзей, которые все еще никак не могут понять, как я нахожу столько времени для писательства. Все они заслуживают благодарности за поддержку моей потребности в написании книг».

Дин: «Выражаю свою благодарности Марку Таберу за то, что он подключил меня к этому проекту. Свою профессиональную карьеру я начал именно с издательства книг по компьютерным технологиям, и мне доставило много удовольствия мое возвращение в эту сферу после десятилетнего перерыва. Я хотел бы поблагодарить Грега Перри за написание первого и второго замечательных изданий, ставших основой для этой версии книги. Для меня было честью работать с ним в качестве редактора первых двух изданий, но еще большей честью для меня стало соавторство при написании нового издания книги. Я лишь могу надеяться, что мое участие было оправданным. Я высоко ценю отличную работу редакторского коллектива Менди Франк, Кристы Хансинг и производственную команду в Пирсон, принявших участие в создании этой книги.

Лично я бы хотел поблагодарить трех своих детей: Джона, Алис и Мегги, а также мою жену Фрэн за их бесконечное терпение и поддержку».

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

#define, 79
#include, 78

break, 167

case, 176
ceil(), 208
continue, 170

do...while, 152

fabs(), 209
fgets(), 306
floor(), 208
fopen(), 301
for, 157
fprintf(), 303
fseek(), 312

getch(), 195
getchar(), 189
gets(), 204

isalpha(), 198
isdigit(), 198
islower(), 198
isupper(), 198

pow(), 209
putchar(), 189
puts(), 204

rand(), 213
return, 339

sqrt(), 209
srand(), 214
strcat(), 203
strcpy(), 223
strlen(), 190
struct, 291
switch, 175

tolower(), 202
toupper(), 202

while, 149

Адреса памяти, 249
Аргумент
  передача по адресу, 333
  передача по значению, 331
Арифметические операции, 94

Бинарные состояния, 20
Бинарный числовой формат, 20

Ввод данных с клавиатуры, 85
Виды данных, 29
Включение файлов в программу, 75
Возврат значений из функции, 339
Вывод строк, 47
Выражение break, 167
Выражение continue, 170
Выражение struct, 291

Глобальная переменная, 325

Декремент значения переменной, 107
Деление по модулю, 96
Директива препроцессора, 75
Директива препроцессора #define, 79
Директива препроцессора #include, 78
Длина строки, 67

Заголовочные файлы, 78

Инициализация строк, 71
Инкремент значения переменной, 107
Исходный код, 19

Команда слияния файлов, 76
Комментарии, 37
Компилятор, 16
Константа, 79

Логарифмические функции, 211
Логические операторы, 125
  порядок выполнения, 134
Локальная переменная, 325

Массив указателей, 265
Массивы
  запись значений, 223
Машинный язык, 19

Нуль-символ, 65

Обслуживание программы, 38
Оператор ветвления if, 116
Оператор ветвления if ... else, 120
Оператор декремента, 142
Оператор инкремента, 142

Оператором структурного указателя
  запись данных в структуру, 293
Оператор разыменования, 252
Оператор-точка
  запись данных в структуру, 293
Оператор условия, 138
Операторы сравнения, 115
Отладка программы, 21
Отступы, 42

Передача аргумента по адресу, 333
Передача аргумента по значению, 331
Переменная, 56
переменная-счетчик, 105
Переменные
  вид, тип, 56
  имя переменной, 58
  объявление, 59
  присвоение значений, 60
Печать строк, 47
Поиск данных, 242
Порядок выполнения операций, 99
Последовательные файл
  модификатор доступа, 302
Последовательные файлы, 300
Приведение типа
  операция, 111
Программа
  определение, 15
Прототип функции, 343
Пузырьковая сортировка, 237

Разыменование указателя, 252

Символ, 30
Символ конца строки, 65
Символы преобразования, 51
Символьные массивы, 68
Случайные числа
  генерация, 213
Сортировка данных, 236
  пузырьковая сортировка, 237
Составной оператор присваивания, 105
Составные операторы, 107
  перечень, 109
  порядок выполнения, 110
  сложение, 108
  умножение, 108
Составные операторы сравнения, 126
Структура, 287

Тригонометрические функции, 210

Указатели, 249
Указатель на файл, 312
Управляющие последовательности, 48
Условный оператор, 138

Файл произвольного доступа, 311
  перемещение по файлу, 312
Файл произвольного досутпа
  модификаторы режимов доступа, 311
Функция
  определение термина, 320
Функция ceil(), 208
Функция fabs(), 209
Функция fgets(), 306
Функция floor(), 208
Функция fopen(), 300, 301
Функция fprintf(), 303
Функция fseek(), 312
Функция getch(), 195
Функция getchar(), 189
Функция gets(), 204
Функция isalpha(), 198
Функция isdigit(), 198
Функция islower(), 198
Функция isupper(), 198
Функция main(), 27
Функция pow(), 209
Функция printf(), 45
Функция putchar(), 189
Функция puts(), 204
Функция rand(), 214
Функция scanf(), 85
Функция sizeof(), 146
Функция sqrt(), 209
Функция srand(), 214
Функция strcat(), 203
Функция strcpy(), 223
Функция strlen(), 190
Функция tolower(), 202
Функция toupper(), 202

Целые числа, 32
Цикл, 148
  бесконечный цикл, 148
Цикл do...while, 152
Цикл for, 156
Цикл while, 149
Цикл ДЛЯ, 156
Цикл ПОКА, 149

Числа с плавающей точкой (запятой),
  32

Чтобы писать мощные программы на C, необязательно быть экспертом! Эта книга максимально быстро поможет вам освоить язык C благодаря невероятно четкому и простому изложению материала. Вы изучите все основные темы, связанные с этим языком: как организовать программу, хранить и отображать данные, работать с переменными, операторами, вводом/выводом, указателями, массивами, функциями и многими другими вещами. Язык программирования C еще никогда не был таким простым!

Оказывается, язык программирования C может быть простым!

Перед вами – лучшая из современных книг по программированию на C для начинающих. Она поможет вам приобрести практические навыки, которые пригодятся при программировании на любом языке. Простые и дельные примеры помогут вам начать создавать различные программы – от игр до мобильных приложений – на языке C.

Самое важное:

- **Структура программы**
- **Логические операторы и выражения**
- **Переменные**
- **Циклы**
- **Встроенные функции**
- **Массивы и указатели**
- **Тестирование программ**
- **Генерация вывода**

и многое другое

Грег Перри обучил программированию тысячи студентов. Он написал более 75 книг по информатике и вычислительной технике, проданных во всем мире общим тиражом более 2 миллионов экземпляров.

Дин Миллер – автор и редактор с более чем 20-летним опытом в издательском бизнесе

ISBN 978-5-699-75873-9

Замечательная книга, полезная не только для изучающих язык C, но и для новичков в программировании вообще. Она знакомит с базовыми понятиями и приемами программирования простым и доступным языком. Большим плюсом является наличие практических примеров.

Александр Ионов, программист