

Быстро освоите основы популярного языка
программирования



С ДЛЯ "ЧАЙНИКОВ"™

2-е издание

**Для
сомневающих**

**Сразу начинайте
программировать
на С —
опыт
не нужен!**

Дэн Гукин

Автор бестселлера
ПК для "чайников"



Д АДАТЕХНИКА

C FOR DUMMIES®

2nd edition

by Dan Gookin



WILEY

Wiley Publishing, Inc.



Дэн Гукин



ДИАЛЕКТИКА

Москва ♦ Санкт-Петербург ♦ Киев
2006

Компьютерное издательство "Диалектика"

Главный редактор *С.Н. Тригуб*

Зав. редакцией *В.Р. Гинзбург*

Перевод с английского и редакция канд. техн. наук *Я.К. Шмидского*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:

info@dialektika.com, <http://www.dialektika.com>

115419, Москва, а/я 783, 03150, Киев, а/я 152.

Гукин, Дэн.

Г93 С для "чайников", 2-е издание. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2006. — 352 с. : ил. — Парал. тит. англ.

ISBN 5-8459-0810-8 (рус.)

В данной книге представлены все наиболее важные сведения о языке C: основные понятия и концепции, наборы символов, ключевые слова, описания и типы переменных, логические выражения, операторы, циклы, основные директивы препроцессора, методика написания (и выполнения) простейших программ. Подробно рассматриваются все этапы подготовки и тестирования программ, написанных на языке C. Все теоретические положения детально демонстрируются на коротких, выразительных примерах.

Книга написана доступным, простым языком. Рассчитана на всех желающих освоить язык C, в том числе и начинающих, не имеющих опыта программирования.

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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 издательства JOHN WILEY & Sons, Inc.

Copyright © 2006 by Dialektika Computer Publishing.

Original English language edition Copyright © 2004 by Wiley Publishing, Inc.

All rights reserved including the right of reproduction in whole or in part in any form. This translation is published by arrangement with Wiley Publishing, Inc.

ISBN 5-8459-0810-8 (рус.)

ISBN 0-7645-7068-4 (англ.)

© Компьютерное изд-во "Диалектика", 2006
перевод, оформление, макетирование

© Wiley Publishing, Inc., 2004

Оглавление

Введение	14
Часть I. Введение в программирование на C	19
Глава 1. Основы языка C	21
Глава 2. Ошибки в программах на C	31
Глава 3. Формальное знакомство с языком C	39
Глава 4. Что такое ввод-вывод?	49
Глава 5. Комментарии: C или не C	63
Глава 6. Ввод-вывод с помощью функций gets() и puts()	71
Часть II. Переменные и некоторая доля математики	77
Глава 7. A + B = C	79
Глава 8. Переменные в языке C	95
Глава 9. Числа в языке C	107
Глава 10. Переменные типа char	119
Часть III. Как научить программы принимать решения	127
Глава 11. Больше математики и Священный порядок (старшинство операций)	129
Глава 12. Могущественная команда if	141
Глава 13. Сравнение символов с помощью ключевого слова if	157
Глава 14. Логические выражения и ключевое слово if	165
Глава 15. Циклы в языке C	173
Глава 16. Знакомство с циклами и применением инкремента (оператора ++) в циклах	187
Глава 17. Познакомьтесь с циклом while (циклом с условием продолжения)	197
Глава 18. Циклы с условием продолжения. Организация задержки	205
Глава 19. Разбор случаев в языке C: переключатель switch-case	217
Часть IV. Язык C: следующий уровень	229
Глава 20. Создание первой функции	231
Глава 21. Переменные в функциях	243
Глава 22. Как на самом деле функционируют функции	251
Глава 23. То, что пишется в начале программы	267
Глава 24. Глава о функции printf()	277
Глава 25. Математическое безумие!	283
Глава 26. Старая функция генерации случайных чисел	293
Часть V. Великолепные десятки	303
Глава 27. Еще десять трюков в языке C	305
Глава 28. Десять подсказок для подающего надежды программиста	311
Глава 29. Десять способов самостоятельно разрешить свои проблемы в программах	317
Приложение А. Прежде чем вы приступите к программированию	323
Приложение Б. Таблица ASCII	333
Предметный указатель	337

Содержание

Введение	14
Часть I. Введение в программирование на C	19
Глава 1. Основы языка C	21
Чрезвычайно короткая и упрощенная хронология языка C	21
Цикл разработки программы на C	22
От текстового файла к программе	23
Исходный текст (текстовый файл)	24
Создание файла исходного текста GOODBYE.C	24
Компилятор и компоновщик	27
Компиляция файла GOODBYE.C	28
Выполнение машинной программы	29
Сохраните исходный текст! Скомпилируйте и скомпонуйте программу! Выполните ее!	29
Глава 2. Ошибки в программах на C	31
Редактирование и перетрансляция — горькие слезы	31
Очередное редактирование файла исходного текста	32
Перетрансляция	32
Горе от ошибок	33
Поиск причины ошибки	34
Исправление ошибочной программы	35
А теперь попробуйте исправить эту ошибку!	36
Глава 3. Формальное знакомство с языком C	39
Большой рисунок	39
Составные части программы на языке C	40
Лексика языка C: ключевые слова	42
Другие компоненты языка C	43
Викторина!	44
Полезная программа RULES (ПРАВИЛА)	45
Важность наличия \n	46
Разбить строки символом \ очень просто	47
Глава 4. Что такое ввод-вывод?	49
Представьтесь компьютеру	49
Компиляция файла WHORU.C	50
Награда	50
Подробнее о printf()	51
Печатаем прикольный текст	51
Как обмануть printf(), или escape-последовательности	53
f означает “formatted” (“форматированный”)	54
Немного выравнивания	55
scanf читается “скан-эф”	57
Компилируем scanf	57
Чудо scanf()	59
Время поэкспериментировать!	59

Глава 5. Комментарии: С или не С	63
Добавление комментариев	63
Большая, запутанная программа с комментариями	64
Зачем нужны комментарии?	65
Стили комментариев: профессиональный и будущих профессионалов (пока еще любителей)	66
Суперкомментарии	66
Комментарии в C++	67
Комментарии как средство отключения кода	68
Опасности вложения комментариев	68
Глава 6. Ввод-вывод с помощью функций gets() и puts()	71
Функция gets(): чем больше я хочу, тем больше я получу	71
Пример программы	72
Недостатки функции gets()	72
Достоинства функции puts()	73
Приглашение к вводу команды: запускаем еще одну глупую программу	73
Функции puts() и gets() в действии	74
Еще одна модификация программы INSULT	74
Функция puts() может печатать переменные	75
Часть II. Переменные и некоторая доля математики	77
Глава 7. A + B = C	79
Вечно изменяющаяся переменная	79
Изменение (замена) строк	80
Выполнение программы KITTY (КОТЕНОК)	80
Добро пожаловать в Холодный Мир Числовых Переменных	81
Привет, целое число	81
Использование целой переменной в программе Methuselah	82
Присваивание значений числовым переменным	83
Ввод числовых значений с клавиатуры	84
Функция atoi()	84
Так сколько же лет было Мафусаилу?	85
А сколько лет вам?	87
Крошечный бит математики	88
Основные математические символы	89
Сколько еще вы должны прожить, чтобы побить рекорд Мафусаила?	90
Дополнительная модификация заключительной версии программы METHUS	91
Прямой результат	92
Глава 8. Переменные в языке C	95
Обсуждение, именование и объявление переменных	95
Зачем объявлять переменные	96
Запрещенные и допустимые имена переменных	96
Предварительная установка значений переменных	97
Старая случайная программа с переменными	99
Возможно, вы хотите еще две пинты?	100
Множественные объявления	101
Константы и переменные	101
Придумывание и определение констант	102
Удобное сокращение	103
Директива #define	104
Настоящие постоянные переменные в действии	105

Глава 9. Числа в языке C	107
Числа, числа и числа, которым несть числа	107
Числа в C	108
Зачем использовать целые числа? Почему нельзя ограничиться только числами с плавающей запятой?	109
Целочисленные типы (короткий — short, длинный — long, широкий — wide, жирный — fat и т.д.)	109
Со знаком или без знака (unsigned), или может ли перед этим стоять знак “минус”?	110
Как объявить число с плавающей запятой	112
Давайте напишем программу, обрабатывающую числа с плавающей запятой!	113
Экспоненциальное представление (с буквой E)	114
Диапазон чисел с плавающей точкой удвоенной точности (double) — еще больше, чем у чисел с плавающей точкой одинарной точности (float)!	116
Форматирование нулей и других знаков десятичного числа	117
Глава 10. Переменные типа char	119
Еще один тип переменной: char	119
Односимвольные переменные: переменные, в которых хранится один символ	119
Символ в действии	121
Запихивая символы в символьные переменные...	121
Чтение и запись отдельных символов	122
Функция getchar()	123
Функция putchar()	123
Значения символьных переменных	124
Часть III. Как научить программы принимать решения	127
Глава 11. Больше математики и Священный порядок (старшинство операций)	129
Краткий обзор основных математических операторов языка C	129
Старая программа определения роста	130
Изменение старой программы определения роста	132
Тонкое искусство приращения (добавления единицы)	133
Приращение веса — к ожирению	134
Еще одна программа! (Она может даже понадобиться в жизни)	135
Священный порядок, или старшинство операций	136
Задача из выпускного экзамена по лечению зубов	136
Что же не так, Моя Дорогая Тетушка Салли?	137
Арифметическая путаница с волшебными шариками	139
Круглые скобки могут изменить приоритет операций, определяемый их порядком (старшинством)	139
Глава 12. Могуущественная команда if	141
Если бы только знать, как использовать if...	141
Пример глупой компьютерной программы	142
Ключевое слово if	144
Форматирование условного оператора	147
Окончательное решение проблемы подоходного налога	148
Что делать, если неравенство не выполняется?	150
Охват всех возможностей с помощью ключевого слова else	150
Формат условного оператора с конструкцией else	151
Странный случай комбинации else-if и множественные альтернативные решения	154
Еще одна программа: подумать только, и в самом деле хитрейший джин	155

Глава 13. Сравнение символов с помощью ключевого слова if	157
Мир ключевого слова if без числовых значений	157
Что больше: S или T, \$ или -?	158
Проблема с getchar()	159
Модификация GREATER.C: правильное считывание стандартного ввода	161
Чтение одиночных символов: функции getchar(), getch() и getche()	162
БОЛЬШАЯ проблема в программе GREATER: два алфавита в ASCII — для строчных и прописных букв	162
Еще один пример программы: меню	163
Как с помощью ключевого слова if сравнить две строки	164
Глава 14. Логические выражения и ключевое слово if	165
Устраняем недостатки в логике программ	165
Команда if и логические операции И (And) и ИЛИ (Or)	166
Решение (но не лучшее)	167
Логические выражения — лучшее решение	168
Логические друзья команды if	169
Программа, в которой используется логическая операция AND (И)	171
Глава 15. Циклы в языке C	173
Понятие цикла	173
Циклическое повторение без проблем	174
Для того чтобы сделать что-нибудь много раз, используйте ключевое слово for	176
Пошаговое выполнение программы OUCH.C	177
Забава: считаем до 100	179
Поговорим о циклах!	180
Полезная программа ASCII	181
Остерегайтесь бесконечных циклов!	182
Выход из цикла	184
Ключевое слово break	185
Глава 16. Знакомство с циклами и применением инкремента (оператора ++)	187
Искусство приращения	187
Загадочные символы операторов языка C, том I: оператор приращения (++)	188
Еще один просмотр программы LARDO.C	188
Таинственная практика уменьшения	189
Отсчет в обратном порядке	190
Цикл: отсчет в обратном порядке	191
Загадочные символы операторов языка C, том II: оператор уменьшения (--)	192
Заключительное усовершенствование OLLYOLLY.C	193
Больше безумия приращения	193
Прыгающие циклы!	194
Считаем до 1 000 пятерками	194
Загадочные символы операторов языка C, том III: безумие продолжается	195
Ответы	196
Глава 17. Познакомьтесь с циклом while (циклом с условием продолжения)	197
Цикл while — цикл с условием продолжения	197
Выполняя цикл while часами	198
Ключевое слово while (формальное введение)	199
Выбор типа цикла: цикл с условием продолжения или цикл for	200
Замена неприглядных циклов for(;;) изящными циклами с условием продолжения	201

Гибкость языка C: сокращения в функциональном стиле	202
Платить за работу вперед — все равно, что гнать мертвую лошадь...	204
Глава 18. Циклы с условием продолжения. Организация задержки	205
Обычный и инвертированный циклы с условием продолжения	205
Цикл с условием продолжения do-while	206
Цикл do-while: подробности с условием продолжения	207
Недостаток программы COUNTDOWN.C	208
Всегда проверяйте вводимые числа в цикле с условием продолжения do-while	208
Вложенные циклы и другие глупости	210
Добавление напряженной, драматической задержки к программе COUNTDOWN.C	210
Программа GRID.C, полная вложенных циклов	213
Ключевые слова break и continue	214
Пожалуйста, продолжайте...	214
Ключевое слово continue	215
Глава 19. Разбор случаев в языке C: переключатель switch-case	217
Эти хитрые циклы — переключатели switch-case	217
Переключатель switch-case в программе LOBBY	219
Старый трюк с переключателем switch-case	221
Переключатель switch-case в циклах с условием продолжения while	224
Часть IV. Язык C: следующий уровень	229
Глава 20. Создание первой функции	231
Знакомство с функциями	231
Глупый пример программы, которую вам не придется вводить	231
Как избежать излишних повторений с помощью функций	233
Благородная функция jerk(), позволяющая устранить повторение безобразий в исходном коде	234
Как функция jerk() работает в программе BIGJERK2.C	235
Создание прототипов функций	236
Основные проблемы создания прототипов	236
Трусливый способ избежать проблем создания прототипов	238
Функции — от А до Я	239
Формат функции	239
Именованые функций, или как давать названия (имена) функциям	241
Глава 21. Переменные в функциях	243
Сброс бомб в программе BOMBER для БОМБАРДИРОВЩИКА!	243
Будет ли бомбить программа BOMBER.C с двумя переменными?	244
Важное добавление: некоторая психологическая напряженность	245
Глобальные переменные: совместное использование и всеобщая любовь	247
Создание глобальной переменной	247
Пример глобальной переменной в реальной, действующей программе	248
Глава 22. Как на самом деле функционируют функции	251
Передача значения функции	251
Как передать значение функции	252
Пример (в самый раз!)	253
Как избежать беспорядка в переменных (обязательное чтение)	255
Передача нескольких значений функции	256
Функции, возвращающие значения	258
Кое-что неприятное: правила	258
Наконец, компьютер вычисляет ваш IQ	259

Возврат значения вызывающей функции с помощью ключевого слова return	261
Теперь вы можете разобраться в главной функции main()	262
Премия — программа BONUS.C!	263
Не беспокойтесь о таких мелочах языка C, если вы спешите	265
Глава 23. То, что пишется в начале программы	267
Пожалуйста, не пропустите меня!	267
Конструкция #include	269
Что такое STDIO.H?	270
Пишем свой собственный файл с расширением .h	271
Последнее предупреждение относительно заголовочных файлов	272
Что такое #define	274
Макроопределения (макросы) — эти темы мы даже не обсуждаем	275
Глава 24. Глава о функции printf()	277
Краткий обзор printf()	277
Старая подпрограмма printf() — отображение текста	277
Escape-последовательности в функции printf()	278
Испытание escape-последовательности в роскошной программе с функцией printf()	279
Испытание PRINTFUN	279
Полный формат printf()	280
Символы преобразования, используемые в функции printf()	281
Глава 25. Математическое безумие!	283
Больше о Математике	283
Возводя математические проблемы в более высокую степень	284
Использование функции pow()	285
Извлечение корня	286
Странная математика!	288
Действительно странные конструкции: декремент и инкремент	289
Опасности использования ++	289
То же самое относится и к --	290
Странное явление отражения ++ от переменной: ++a	291
Глава 26. Старая функция генерации случайных чисел	293
Случайность	293
Функция rand()	294
Инициализация генератора случайных чисел	295
Повышаем непредсказуемость программы RANDOM	296
Упрощение генератора случайных чисел	298
Дьявольский доктор Модуль	299
Окончательная версия программы RANDOM; СЛУЧАЙНЫМ образом бросаем игральный кубик	301
Часть V. Великолепные десятки	303
Глава 27. Еще десять трюков в языке C	305
Массивы	305
Строки	306
Структуры	307
Указатели	308
Связные списки	308
Двоничные операторы	308
Использование параметров командной строки	309
Доступ к файлам на диске	310

Взаимодействие с операционной системой	310
Разработка больших программ	310
Глава 28. Десять подсказок для подающего надежды программиста	311
Команда history (хронология) позволяет использовать хронологию командной строки	311
Держите ваш редактор открытым в другом окне	312
Используйте текстовый редактор с цветной раскраской контекста	312
Используйте в вашем редакторе команды, в которых применяются номера строк	313
Держите открытым окно приглашения к вводу команды, если используете интегрированную среду разработки	314
Несколько удобных команд в окне с приглашением к вводу команды	314
Правильно называйте ваши переменные	315
Загадки пост- и предприсваивания и уменьшения	315
Выход из цикла	315
Глава 29. Десять способов самостоятельно разрешить свои проблемы в программах	317
Исправляйте только одну ошибку за один раз	317
Разбивайте код на части	318
Упрощение	318
Обсуждение программы	319
Устанавливайте контрольные точки	319
Контролируйте ваши переменные	320
Документируйте вашу работу	320
Используйте средства отладки	321
Используйте оптимизатор кода на C	321
Читайте книги!	321
Приложение А. Прежде чем вы приступите к программированию	323
Настройка и установка	323
Компилятор с языка C	323
Место для хранения проектов	324
Создание программ	326
Поиск каталога (папки) leapt	326
Запуск редактора	327
Компиляция и редактирование связей	328
Приложение Б. Таблица ASCII	333
Предметный указатель	337

Об авторе

Дэн Гукин пишет о технических и прикладных науках уже 20 лет. Он написал для многих журналов немало статей по высоким технологиям и более 90 книг по технологии вычислений на персональном компьютере, многие из которых исчерпывающие.

Совмещая увлечение техническими и прикладными науками с любовью к написанию книг, он создает шедевры, которые являются информативными и при этом занимательны, а не скучны. Продав более 14 миллионов книг на более чем 30 языках, Дэн доказал, что его метод создания компьютерных фолиантов действительно работает.

Возможно, самой известной книгой Дэна является первая книга *DOS For Dummies* (DOS для “чайников”), опубликованная в 1991 году. Она стала самой быстро продаваемой компьютерной книгой, причем количество экземпляров этой книги, реализованных за неделю, превосходило аналогичный показатель для бестселлера номер один по версии “Нью-Йорк Таймс” (правда, поскольку эта книга — справочник, она не могла быть включена в список бестселлеров по версии Нью-Йорк Таймс).

Последние книги Дэна включают в себя *PCs For Dummies* (ПК для “чайников”), 9-е издание; *Buying a Computer For Dummies* (Покупка компьютера для “чайников”), издание 2004 года; *Troubleshooting Your PC For Dummies* (Устранение неисправностей в ПК для “чайников”); *Dan Gookin's Naked Windows XP* (Все о Windows XP от Дэна Гукина) и *Dan Gookin's Naked Office* (Все об Office от Дэна Гукина). Он также издает бесплатный еженедельный информационный бюллетень *Weekly Wambooli Salad*, полный подсказок и компьютерных новостей. А кроме того, Дэн ведет обширную и весьма полезную Web-страничку www.wambooli.com.

Диплом по технике связи и изобразительному искусству Дэн получил в Калифорнийском университете в Сан-Диего. Он живет на северо-западе США, где вместе со своими четырьмя сыновьями наслаждается прогулками по тихим лесам Айдахо.

Введение

Добро пожаловать во 2-е издание книги *С для чайников* — вашу самую последнюю, решительную и победную попытку понять язык программирования С.

Хотя я и не могу обещать, что после преодоления этого текста вы станете профессионалом в С, но я могу гарантировать, что вы:

- ✓ сумеете найти программу на С даже в кипе бланков налоговых служб, утренних биржевых отчетов, бейсбольных статистик и еще Бог знает каких документов, написанных шрифтом Брайля, — из всего этого вы безошибочно сможете выбрать то, что является программой на С;
- ✓ сможете писать такие программы на С, которые никакой другой издатель не разрешил бы автору напечатать в книгах по языку С;
- ✓ сможете оценить остроумие следующего кода, хотя и не сможете с его помощью произвести впечатление на друзей во время вечеринки:
пока (за работу платят вперед)
 бить_баклуши();
- ✓ научитесь разговаривать на языке С, что является способностью читать наборы букв, такие как `printf`, `putchar` и `clock`, и произносить их как “принт-эф”, “пут-кар” и “си-лок”;
- ✓ получите огромное удовольствие.

На самом деле я, конечно, не могу гарантировать последний пункт. Однако я могу гарантировать, что при чтении этой книги над вашей головой не будет висеть Дамоклов меч, угрожающе раскачивающийся при каждой вашей математической ошибке. Давайте оставим строгое программирование тем, кого волнуют последовательности Фибоначчи, число Авогадро, а также преимущества той или иной системы форматирования исходных программ на С с помощью отступов. Серьезная работа — для зануд. А веселье случается, когда вы читаете 2-е издание книги *С для “чайников”*.

Зачем изучать С?

Взгляните на экран компьютера. Представьте, что там что-то происходит. Что угодно. Когда вы научитесь программировать компьютер, в нем будет происходить именно то, что вы захотите. Ладно, может не так скоро, как вам бы того хотелось, но этого можно достичь.

Программирование — это окончательный способ расквитаться с компьютером. *Вы* будете им руководить. *Вы* будете приказывать этому упрямцу, что делать. И он будет подчиняться вам, даже если вы заставите делать его какую-нибудь глупость. Компьютеры быстры и послушны, но не умны.

Всем, что делает компьютер, любыми устройствами, которые находятся внутри него или подключены к нему, можно управлять, кодируя программы на языке программирования, — именно эти программы будут тянуть за нужные рычаги и нажимать нужные кнопки. Язык программирования С — наилучшее (и самое распространенное) средство программирования любого персонального компьютера. Может быть, язык программирования С и не самый простой для понимания, но, уж точно, не самый сложный. Он потрясающе популярен и хорошо поддерживается на всех платформах, именно поэтому лучше всего изучить именно его.

О бесподобном методе изучения "Смотри сюда, "чайники" "

Большинство книг о программировании начинаются с предположения, что вы ничего не знаете. Но авторы об этом помнят на протяжении двух или, в лучшем случае, трех глав. Далее первоначальное стремление писать для новичков испаряется, и тут они начинают! Главу 4 такие авторы пишут не для того, чтобы научить вас программировать, а чтобы произвести впечатление (наверное, правильное было бы сказать огоршить) на своих приятелей-программистов из колледжа. И в результате этого ваше путешествие в мир программирования заканчивается слезами. Но это, безусловно не относится к данной книге.

Самый лучший способ изучать что-либо — изучать это по частям, по одной определенной части за один раз. В программировании я предпочитаю демонстрировать конструкции и приемы, используя короткие программы, небольшие модели и примеры, которые можно быстро ввести в компьютер. В моих программах вы не найдете инициализирующей части, код которой растянется на три страницы; и потому вы не заблудитесь в программе, даже если прочтете еще несколько глав, поскольку темп остается неизменным на протяжении книги. Именно так я написал данную книгу!

В этой книге вы сразу же приступаете к делу. Проведя статистические исследования других книг, я обнаружил, что зачастую первая программа, которую читатель должен ввести в компьютер, находится примерно на пятидесятой странице книги и притом занимает несколько десятков строк! В этой книге первый пример программы появится уже в середине весьма коротенькой главы 1 "Основы языка C". Так быстро!

Как изучать примеры из этой книги

Часть удовольствия от изучения программирования при помощи книги состоит именно в самостоятельном написании программ. Именно так я научился программировать компьютер. Я взялся за дело с книгой *Learning TRS-80 BASIC* (Изучаем TRS-80 BASIC) Дэвида Лайена (David Lien), выпущенной издательством "Compusoft", и, спустя целых 36 часов, я закончил. Потом я спал. Потом проделал все снова, так как совершенно все забыл, но помнил, что мне понравилось делать это в первый раз. Ваше первое задание — прочесть приложение А "Прежде чем вы приступите к программированию". В нем рассказывается, как установить компилятор языка C на ваш компьютер и подготовиться к работе.

Затем вам надо знать, как печатать текст программы. Он может выглядеть следующим образом:

Вот я начинаю печатать материал. Ля, ля, ля.

В основном вы будете писать полные программы, состоящие из нескольких строк, таких как строка перед этим абзацем. Печатайте их полностью, а потом нажимайте Enter в конце каждой строки. Однако из-за того, что ширина этой книги ограничена, иногда будут встречаться строки, разбитые на две части. Это выглядит так:

Вот пример очень длинной строки, которая была мучительно разбита на две жестокими наборщиками этой книги.

Когда увидите такое, не печатайте две строки. Продолжайте печатать, на экране все поместится в одну строку. Если забудете этот совет, программы будут испорчены, так что я периодически напоминаю об этом.

Тупые предположения

Когда я писал эту книгу, я кое-что предполагал о вас, вашем компьютере, вашем компиляторе и, что самое главное, о вашем умонстроении:

- ✓ у вас есть компьютер, или, по крайней мере, у вас есть доступ к нему. Это касается любого компьютера, эта книга предназначена *не только* для программирования под Windows;
- ✓ вы не боитесь компьютера. Вы разбираетесь в нем и даже можете устранить неполадки или помочь другим решить их проблемы;
- ✓ вы умеете пользоваться Internet и загружать оттуда то, что вам нужно. Кроме того, вы знаете, как пользоваться поисковыми машинами;
- ✓ вы хотя бы приблизительно знакомы с интерфейсом командной строки и окнами на вашем мониторе. Это важно, и в приложении А “Прежде чем вы приступите к программированию” объясняется, почему;
- ✓ вы хотите научиться программировать, возможно даже, что вы отчаянно этого хотите!

Пиктограммы, используемые в этой книге



Технические подробности, которые можете с удовольствием пропустить.



Что нужно не забывать делать.



Помните, что этого делать не следует никогда.



Полезный совет, стоящий внимания.

Что нового в этом издании?

На самом деле эта книга не является вторым изданием никакой предыдущей книги, но она использует материал старых книг *C For Dummies* (*С для “чайников”*), тома I и II. Эта книга представляет собой объединение основного материала из обеих книг. И, прочитав эту книгу, вы обретете обширные фундаментальные познания в языке С.

В отличие от упомянутых ранее старых книг, эта книга организована по главам, а не по урокам. Каждая глава является автономной, но, где необходимо, даны перекрестные ссылки на другие главы.

Больше нет викторин и тестов. В этой книге домашние задания, по существу, отсутствуют.

Увы, объем этой книги ограничен, материал излагается неспешно и детально, и потому некоторые сложные аспекты языка не рассмотрены. Так что те из читателей, кто захочет про-

должить изучение языка С, должны будут обратиться к сопровождающей книге *C All-in-One Desk Reference For Dummies*, выпущенной издательством “Wiley”. Эта книга предназначена для более опытных программистов, вы станете одним из них, после того как ее прочтете.

Заключительные размышления

Изучение технологии программирования на С — непрерывный процесс. Только глупец может сказать: “О программировании на языке С я знаю все”. Каждый день для изучения появляется новый материал и другие подходы к одним и тем же задачам. Ничто не совершенно, но многие вещи близки к совершенству.

Мои рассуждения по данной теме таковы: “Конечно, люди, которые занимались программированием на С 20 лет и слишком много платили за семестр в специализированном университете, будут обладать некоторым снобизмом. Неважно. Задайте себе такой вопрос: “Моя программа работает? Хорошо. Делает ли она то, что я хочу? Еще лучше! Отвечает ли она их искусственным стандартам? Какая разница?” Я буду счастлив, если ваша несколько сыроватая программа будет работать. Но помните: чем больше вы учитесь, тем лучше вы становитесь. Вы будете открывать новые приемы и приспосабливать ваш стиль программирования под них.”

У этой книги есть сопровождающая Web-страница, снабженная дополнительным материалом и всевозможной забавной информацией: <http://www.c-for-dummies.com>.

Я надеюсь, что вам понравится путешествие, которое вы намереваетесь начать. Закатайте рукава, запустите компилятор и будьте готовы несколько битых часов неотрывно следить за процессом. Вы собираетесь программировать на С!

Ждем ваших отзывов!

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

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

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

E-mail: info@williamspublishing.com
WWW: <http://www.williamspublishing.com>

Адреса для писем:

из России: 115419, Москва, а/я 783
из Украины: 03150, Киев, а/я 152

Часть I

Введение в программирование на C



"Мы пришли почистить коз!"

В этой части...

Вы никогда ничего не программировали в своей жизни? А видеомэгни-тофон? Забыли о нем?! А на микроволновой печи вы ведь нажимаете кнопки *Попкорн* и *Добавить минуту*. К тому же вы знаете, что можете постучать по клавишам мобильного телефона, чтобы ввести цифры, а затем нажать кнопку *Отправить*, но все же не осмеливаетесь прикоснуться к каким-либо другим клавишам, опасаясь *программирования*, будто оно ведет вас в темное царство с его сырыми заплесневевшими темницами. Если это так, приготовьтесь изменить вашу жизнь к лучшему.

Вопреки всему тому, что вы можете думать, программировать компьютер — суший пустяк. Любой может делать это. Программисты просто иногда создают вокруг себя атмосферу мистики и используют свои профессиональные навыки подобно жрецам, исполняющим священные религиозные обряды. Вздор. Программировать безболезненно. Это легко. Это весело.

Теперь ваша очередь приказывать компьютеру, что именно ему нужно с собой делать. Всего через несколько страниц вы будете программировать ваш ПК. Время расквитаться! Время скрутить ему руки и ждать, пока он не завопит: "Дядя! ДЯДЕНЬКА!". Готовьтесь руководить им!

Основы языка C

В этой главе...

- Историчная хронология C
- Как создать программу на C
- Формирование исходного текста
- Компиляция и компоновка
- Выполнение программы

Компьютер — самое полезное устройство из всех когда-либо использованных вами, ведь он может стать чем угодно, если вы умеете программировать. Именно это свойство компьютеров делает их уникальными в пантеоне современных устройств. И хотя большинство компьютерных пользователей уклоняется от программирования, путая программирование с математикой или разработкой электрических схем, на самом деле программировать компьютер довольно просто и никакой магии для этого не нужно. Это совсем легко.

В этой главе обсуждаются основы программирования. Хотя программирование содержит некоторую долю философии и имеет свои основы, суть главы состоит в том, чтобы вместе с вами создать, скомпилировать и выполнить вашу первую программу. Почувствуйте власть над компьютером! Наконец, именно вы будете приказывать компьютеру, что ему делать!



Поскольку, вероятно, вы не читали введение к этой книге (какой позор!), замечу, что предварительно вы должны просмотреть приложение А “Прежде чем вы приступите к программированию” перед тем, как продолжите чтение этой главы.

Чрезвычайно короткая и упрощенная хронология языка C

Сначала был язык программирования В. Потом был язык программирования С.

Нет, я ничего не перепутал. Язык С был разработан в лаборатории AT&T Bell Labs в начале 1970-х. В то время в лаборатории Bell использовали язык программирования по имени В — В в честь Bell. Затем на основе В был создан следующий язык — С.

- ✓ Язык С — потомок языка программирования В и языка BCPL (Basic Combined Programming Language — Основной Объединенный Язык Программирования, некий машинно-независимый язык системного программирования). Нужно признать, что история языка В достаточно интересна сама по себе и ее лучше изложить где-нибудь в другом месте.
- ✓ Возможно, вы подумали, что следующую, лучшую версию С назовут языком D. Но это не так; ее назвали C++ по причинам, которые станут очевидными в главе 16 “Знакомство с циклами и применением инкремента (оператора ++ в циклах”.



- ✓ Язык С считается языком среднего уровня. Скучные подробности о том, что это значит, содержатся во врезке “Уровень языка (это знать совсем не обязательно)”.
- ✓ Язык программирования С в лаборатории Bell Labs создал Деннис Ритчи. Я упоминаю об этом на тот случай, если вы когда-нибудь на вечеринке захотите похвастаться перед своими друзьями. В этом случае вы можете сказать: “А ведь знаете, С изобрел Деннис Ритчи”. И после некоторого раздумья, напустив туману, можете добавить: “Это было круто”.



Уровень языка (это знать совсем не обязательно)

В зависимости от того, насколько языки программирования напоминают человеческие языки, они относятся к различным уровням. Языки программирования, в которых используются обычные слова и которым относительно просто для большинства людей научиться читать и понимать их, называют языками высокого уровня. Противоположность им — языки низкого уровня, научиться читать и понимать их совсем непросто.

К языкам высокого уровня относится популярный язык программирования BASIC (БЕЙСИК), а также другие языки, которые уже не так популярны. Читать программы на BASIC (БЕЙСИК) почти так же просто, как романы на английском языке, и все его команды и инструкции — английские слова, которые написаны полностью или в которых пропущено несколько гласных либо просто не соблюдаются правила правописания.

Самый низкий уровень из всех языков программирования нижнего уровня имеет машинный язык. Этот язык фактически представляет собой запись примитивного мычания и ворчания самого микропроцессора. Машинный язык состоит из чисел и кодов, которые микропроцессор понимает и выполняет. Но на самом деле никто не пишет программы на машинном языке. Программисты при необходимости используют ассемблер, который на одну ступеньку выше машинного языка (языка самого низкого уровня), потому что мычание и ворчание представлено на нем более понятно, а не в виде чисел.

Почему же иногда используется язык нижнего уровня, хотя существуют языки высокого уровня? По причине скорости! Программы, написанные на языках низкого уровня, выполняются с такой скоростью, с какой компьютер может выполнить их, т.е. часто во много раз быстрее, чем программы, написанные на языках высокого уровня. Кроме того, размер программы на ассемблере обычно меньше. Программа, написанная в Visual Basic, может занимать 34 Кбайт памяти, но та же самая программа, написанная в ассемблере, может занимать всего лишь 896 байт. С другой стороны, время, которое требуется, чтобы разработать программу на языке ассемблера, намного превосходит то, которое требуется для записи той же самой программы на языке высокого уровня. Это компромисс.

Язык программирования С считают языком среднего уровня. Кое-что в нем похоже на хрюканье и ворчание нижнего уровня, но очень много такого, что относится к языку высокого уровня, который читается подобно любому предложению из романа Майкла Крайтона, но, конечно, в разработку программ нужно вложить больше души. Язык С объединяет лучшие свойства языков программирования высокого уровня и скорость разработки программ, которая достижима только для них, с небольшим размером программы и скоростью выполнения, присущей языкам низкого уровня. Именно поэтому все так обожают С.

Цикл разработки программы на С

Вот как создается программа на С — за семь шагов. (Эти шаги называются циклом разработки).

1. Придумать идею программы.
2. С помощью редактора записать исходный текст.

3. С помощью компилятора языка С скомпилировать исходный текст, а затем скомпоновать программу.
4. Горько плакать над ошибками (не обязательно).
5. Выполнить программу и проверить ее.
6. Рвать волосы из-за ошибок (не обязательно).
7. Начать все с начала (вот это уж обязательно).

Нет никакой необходимости запоминать этот список. Это похоже на инструкции на бутылке шампуня, хотя вы и не должны обливаться водой во время программирования на компьютере. В конечном счете, все выйдет точно так же, как мытье шампунем. Вы выполняете инструкции, не думая об этом. Совсем нет никакой необходимости запоминать что-либо.

- ✓ Цикл разработки программ на С — это не тренажер.
- ✓ Шаг 1 самый тяжелый. Все остальное выполняется естественно.
- ✓ Шаг 3 состоит из двух шагов: компиляции и компоновки. В большей части этой книги я рассматриваю их вместе, в одном шаге. Только позже, если вы все еще сохраните интерес, я объясню различия между компилятором и компоновщиком.

От текстового файла к программе

Когда вы создаете программу, вы становитесь программистом. Ваши друзья или родственники могут называть вас “компьютерным мастером” или “гуру”, но поверьте мне, что программист — намного более точное название.

Задача программиста, конечно, заключается в “программировании”, но, если хотите произвести впечатление, лучше употребляйте более вычурный термин: кодирование (программы). Поэтому то, что вы делаете, когда садитесь писать программу, — кодирование программы. Привыкайте к этому термину! Он очень модный.

Задача программиста состоит в том, чтобы написать некоторый код! Код, чтобы сделать что? И какой код вы используете? Секретный код? Код азбуки Морзе? Почтовый индекс?

Цель компьютерной программы состоит в том, чтобы заставить компьютер сделать кое-что.

Цель программирования состоит в том, чтобы “заставить нечто случиться внутри компьютера”. Язык С — только инструмент для связи с ПК. Именно программист должен транслировать (перевести) намерения пользователя компьютера в нечто такое, что компьютер понимает и затем дает пользователям то, что они хотят. И если вы не можете дать им то, что они хотят, заставьте компьютер делать то, что по крайней мере похоже на то, что они хотят, чтобы они не жаловались постоянно, ибо может быть еще хуже — они захотят получить свои деньги назад.

Язык программирования С как раз и есть тот инструмент, с помощью которого вы заставляете некоторые события происходить в компьютере. Это — код, который вы используете, чтобы общаться с ПК. В следующих разделах описано, как все это происходит. В конце концов, сказать “Привет, компьютер!” можно и с помощью мыши.

- ✓ Программирование — это то, что делают составители телевизионных программ. Компьютерные программисты кодируют.
- ✓ Язык программирования позволяет сказать компьютеру, что он должен делать.

Исходный текст (текстовый файл)

Поскольку компьютер не может понять речь, орать на него бесполезно, лучше всего общаться с ним с помощью записки для компьютера — файла на диске.

Чтобы создать послание для ПК, используется программа, называемая текстовым редактором. Эта программа — примитивная версия текстового процессора, из которого выброшены все средства форматирования и поражающие воображение опции печати. Текстовый редактор позволяет напечатать текст — вот почти и все.

С помощью текстового редактора вы создаете то, что называется *файлом исходного текста*, или просто *исходным текстом*. Единственное отличие этого файла состоит в том, что он содержит команды, которые говорят компьютеру, что сделать. Конечно, было бы хорошо, если бы можно было написать команды вроде “Делай забавный шум”. Но горькая правда состоит в том, что команды нужно записать на языке, который компьютер понимает. Например, команды можно написать на языке C.

- ✓ Файл исходного текста — это текстовый файл на диске. Файл содержит команды для компьютера, которые написаны на языке программирования C.
- ✓ Файл исходного текста создается с помощью текстового редактора. О текстовых редакторах более подробно рассказывается в приложении А “Прежде чем вы приступите к программированию”.

Создание файла исходного текста GOODBYE.C

С помощью вашего текстового редактора создайте следующий исходный текст. Тщательно напечатайте каждую строку точно так, как написано; все, что вы видите ниже, важно и необходимо. Ничего не пропустите¹:

```
#include <stdio.h>
int main()
{
    printf("Goodbye, cruel world!\n");
    return(0);
}
```

Почему все программы состоят из английских слов

Вполне возможно, что вам легче было бы ввести программу, если бы в ней слова были не английские, а русские, и потому вы задумываетесь, нельзя ли заменить их русскими. Давайте в этом разберемся.

Сначала выясним, почему все программы состоят из английских слов. Здесь причины исторические. Дело в том, что первые ЭВМ и языки программирования разрабатывались еще в середине прошлого столетия в США, Великобритании и СССР. Поскольку в СССР вся промышленность была подчинена КПСС, которая ставила своей задачей построение коммунизма во всем мире, все разработки велись в рамках военно-промышленного комплекса. И потому именно генералы (от КПСС) определяли, что необходимо советскому человеку. Так, задача облегчения (и, в частности, автоматизации) производственных процессов рассматривалась не как первоочередная, а как подчиненная. КПСС привыкла

¹ Если вас смущает то, что в программе так много английских слов, прочтите врезку “Почему все программы состоят из английских слов”, написанную редактором специально для данного издания. — Прим. ред.

брать не автоматикой, а численностью исполнителей, и за ценой никогда "не стояла". Если во славу вояждя нужно было построить какое-нибудь сооружение, всегда можно было арестовать соответствующее количество "врагов народа". Чтобы повернуть северные реки на юг (такой проект всерьез рассматривался в 80-е годы прошлого столетия, и даже были начаты подготовительные работы), были подготовлены изменения в законодательстве, позволявшие арестовать почти все население, и даже были заказаны наручники. КПСС считала, что и в программировании, как и в рытье каналов, все можно будет взять с помощью "мартышкиного труда". А поскольку языки программирования относились к средствам автоматизации программирования, в СССР их разработкой занимались лишь энтузиасты, которые очень часто даже не имели возможности общаться между собой (ведь все средства информации, включая и научные журналы, контролировались КПСС). Так что не удивительно, что почти всю информацию по разработке и конструированию языков программирования эти энтузиасты черпали из англоязычных источников. Поэтому даже разрабатывавшиеся в СССР языки программирования зачастую были основаны на английской лексике. К тому же КГБ, посулив баснословные суммы, сумел вывезти из тщательно охраняемых хранилищ фирмы IBM семь грузовиков секретнейшей документации по разработке машин серии IBM 360. В СССР по этой документации было развернуто широкомасштабное клонирование машин серии IBM 360, которые были названы ЕС ЭВМ. Когда появились первые модели, выяснилось несколько обстоятельств. Во-первых, стало понятно, что советская промышленность не может сделать полноценных копий даже при наличии полной документации. Во-вторых, стало понятно, что ценность вывезенной секретной документации равна цене макулатуры, поскольку она устарела еще задолго до кражи. В-третьих, стало ясно, что фирма IBM с помощью КГБ захватила весь рынок ЭВМ в СССР и заодно устранила всех потенциальных конкурентов на территории СССР, так как с началом развертывания программы выпуска ЕС ЭВМ финансирование всех остальных научных разработок было прекращено. Ну, и, в-четвертых... американский английский стал родным для всех советских программистов. (Хорошо же фирма IBM хранила свои секреты!) Так что теперь вы знаете, кто виноват в том, что все программы состоят из английских слов.

Теперь разберемся, можно ли в программах вместо английских слов использовать русские. Как вы уже знаете, микропроцессор не понимает ни тех, ни других, так что программы на языках высокого уровня все равно переводятся (программисты говорят транслируются) на язык цифр, понятный микропроцессору. Поэтому может возникнуть вопрос: нельзя ли в таком случае транслировать с русского на английский (или в противоположном направлении)? В принципе можно. В одном из скандинавских университетов в 70-х годах прошлого столетия большой популярностью пользовалась система, которая транслировала программы, написанные английскими словами, на алгоритмический язык Аналитик (этот язык понимали машины МИР-2 и МИР-3, разработанные небольшой группой энтузиастов в СССР). Это давало возможность использовать богатые аналитические возможности машины МИР-2 тем, кто не освоил трудного для них русского языка. Справедливости ради нужно сказать, что, если не считать его предшественника (язык Алмир) и так и не реализованной национальной версии языка Алгол-68, Аналитик был единственным алгоритмическим языком высокого уровня с русской лексикой. История создания этого языка является ярким подтверждением сказанного ранее о негативном отношении КПСС к передовым технологиям: против одного из создателей языка (Боднарчука В.Г.) было сфабриковано политическое обвинение, и, чтобы избежать ареста, он вынужден был сменить место работы и выехать из Киева.

Но, как бы то ни было, нужно отметить два факта. Во-первых, широко доступных систем, транслирующих программы с русского на английский, в настоящее время нет, да и в дальнейшем разработка их не предвидится. Во-вторых, количество английских слов, вошедших в С, весьма ограничено (примерно три десятка в минимальных версиях языка С), запомнить их совсем нетрудно. Кроме того, слова эти не изменяются по падежам, числам, склонениям и т.д. Так что с этими словами проблем нет. Зато гораздо разнообразнее набор слов, которые могут встретиться в строках (это то, что заключено в кавычках). В принципе, в строках могут встречаться любые символы, в том числе и русские буквы. Однако то, как они будут отображаться у вас на экране, зависит от вашей среды программирования и операционной системы. В частности, русские буквы в некоторых средах, если не предпринять специальных мер, могут отображаться в виде букв с разнообразными значками (вроде умляутов и цедил) или вообще каких-нибудь загадочных значков. Поэтому вы можете попробовать использовать в строках русские буквы, но если появятся значки, советую перейти на латинский регистр и все примеры набирать так, как показано в листингах. Для удобства я после

двух косых даю перевод, вы можете даже не набирать его — это комментарий. Комментарий — это то, что не оказывает никакого влияния на программу, он просто игнорируется. Вот как, например, выглядит программа GOODBYE.C с комментариями (они выделены наклонным шрифтом)².

```
#include <stdio.h>
int main()                                // главная функция
{
    // До свидания, жестокий мир!
    printf("Goodbye, cruel world!\n");
    return(0);                            // возврат
}
```

Просматривая напечатанное, обратите внимание, что некоторые слова вам знакомы³. Вы узнаете некоторые слова и фразы (include (включить), main (главный), "Goodbye, cruel world!" ("До свидания, жестокий мир!") и return (возврат)), но некоторые слова выглядят странно (stdio.h, printf и эта совсем странная штука \n).

Когда закончите набирать команды, сохраните их в файле на диске. Назовите файл GOODBYE.C. С помощью команд вашего текстового редактора сохраните этот файл, а затем возвратитесь к приглашению к вводу команды, чтобы скомпилировать введенные команды в программу.

- ✓ В приложении А "Прежде чем вы приступите к программированию" рассказывается, как использовать текстовый редактор для записи программы на С, а также приведены инструкции, где на диске сохранить файл исходного текста.
- ✓ Если вы пользуетесь Блокнотом (Notepad) системы Windows, то нужно убедиться, что имя файла заканчивается на .C, а не на .TXT. Найдите книгу по Windows, в которой описаны команды, необходимые для отображения расширений имен файлов, тогда вам легче будет сохранять текстовый файл на диске с расширением .C.
- ✓ Обратите внимание, что текст набирается главным образом на нижнем регистре. Это важно, потому что многие языки программирования не только чувствительны к регистру, а просто помешаны на соблюдении регистра. Не волнуйтесь, если английская грамматика или правила пунктуации идут вразрез с языком С; С — машинный язык, а не английский язык.
- ✓ Обязательно также обратите внимание на расстановку в программе различных скобок: обычных круглых (и), угловых < и >, а также фигурных { и }.



Дополнительные сведения, которые могут понадобиться при вводе исходного текста GOODBYE.C

Вот первая строка:

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

Напечатайте знак фунта (нажмите <Shift+#>), а затем include и пробел. Напечатайте левую угловую скобку (этот знак на клавиатуре расположен над запятой), а затем stdio, точку, h и правую угловую скобку. Все должно быть на нижнем регистре — никаких заглавных букв! Нажмите <Enter>, чтобы закончить эту строку и начать вторую строку.

² Врезка добавлена редактором русского перевода. — Прим. ред.

³ Если вы учили английский язык. Если нет, прочтите врезку "Почему все программы состоят из английских слов", написанную редактором специально для данного издания. — Прим. ред.

На второй строке нажмите только одну клавишу <Enter> — строка будет пустой. Пустые строки часто используются в кодах программ: они добавляют пространство, которое отделяет части кода и делает его более читаемым. И, поверьте мне, все, что улучшает читаемость кода программы, очень полезно!

Напечатайте слово `int`, пробел, `main` (главный), а затем две круглые скобки, в которых ничего нет. У вас получится строка `int main()`.

Между `main` и круглыми скобками, а также в самих круглых скобках никакого пробела не нужно⁴. Нажмите <Enter>, и тем самым вы перейдете на четвертую строку.

Напечатайте левую фигурную скобку:

```
{
```

Этот символ находится в отдельной строке, в самом начале строки. Нажмите <Enter>, чтобы начать пятую строку.

```
printf("Goodbye, cruel world!\n");
```

("До свидания, жестокий мир!\n")

Хорошо, если ваш редактор догадается автоматически сделать отступ в этой строке. В противном случае чтобы сделать отступ, нажмите клавишу табуляции (<Tab>). Затем напечатайте `printf`, т.е. слово `print` (печать) с маленькой буквой `f` в конце. (Это слово произносится "принт-эф.") Напечатайте левую круглую скобку. Напечатайте двойную кавычку. Напечатайте `Goodbye, cruel world` (До свидания, жестокий мир), а затем — восклицательный знак. После этого напечатайте наклонную черту влево, маленькую букву `n`, двойную кавычку, правую круглую скобку и, наконец, точку с запятой. Нажмите <Enter>, и вы попадете на шестую строку.

```
return(0);
```

(возврат(0);)

Если редактор автоматически не сделает отступ в шестой строке, нажмите клавишу табуляции (<Tab>), чтобы строка начиналась с отступа. Затем напечатайте `return` (возврат), скобку, `0` (нуль), скобку и точку с запятой. Нажмите <Enter>.

На седьмой строке напечатайте правую фигурную скобку:

```
}
```

Некоторые редакторы автоматически выбирают отступ для этой скобки. Если ваш редактор этого не делает автоматически, вручную расположите скобку так, чтобы она была первым символом в строке. Нажмите клавишу <Enter>, чтобы закончить эту строку.

Оставьте восьмую строку пустой.

Компилятор и компоновщик

После того как исходный текст создан и сохранен на диске, его нужно оттранслировать на язык, который может понять компьютер. Эту задачу решает компилятор.

Компилятор — специальная программа, которая читает команды, хранящиеся в файле исходного текста, исследует каждую команду и затем транслирует полученную информацию в машинный код, понятный только микропроцессору компьютера.

⁴ Не беда, если он там окажется. — Прим. ред.

Если все идет хорошо и компилятор удовлетворен вашим исходным текстом⁵, компилятор создаст файл объектного кода. Это — промежуточный шаг; он не является абсолютно необходимым для маленьких программ, но он жизненно необходим для больших программ.

Наконец, компилятор компоует файл объектного кода, в результате чего и получается настоящая работающая компьютерная программа.

Если компилятор или компоновщик чего-то не понимают, они выдают сообщения об ошибках. Вы при этом можете скрежетать зубами, злиться и проклинать все на свете, но чтобы вы ни делали, вам придется снова отредактировать файл исходного текста и устранить все ошибки, найденные компилятором. (Это не столь ужасно, каким кажется на первый взгляд.) Затем попытайтесь откомпилировать программу снова — это действие часто называется перетрансляцией и перекомпоновкой.



- ✓ Компилятор транслирует инструкции, хранящиеся в файле исходного текста, в команды, которые может понять компьютер. Компоновщик затем конвертирует (преобразовывает) эту информацию в исполнимую программу.
- ✓ В компиляторе GCC, который автор горячо рекомендует (сам им пользовался при написании этой книги), шаги компиляции и компоновки объединены. Объектный файл GCC создает, но он его автоматически удаляет после создания файла с окончательной (машинной) программой.
- ✓ Файлы объектного кода имеют расширение OBJ или (иногда) только O. Первая часть названия (имени) объектного файла та же самая, что и имя файла исходного текста.
- ✓ Пока не утруждайте себя запоминанием всей этой ерунды об объектном коде. С радостью можете забыть ее, если она вам мешает.
- ✓ Текстовый редактор ⇨ компилятор.
- ✓ Исходный текст ⇨ программа.

Компиляция файла GOODBYE.C

Технические подробности, касающиеся компиляции программы, вынесены в приложение А “Прежде чем вы приступите к программированию”. Предполагая, что вы уже читали его, прошу напрячь вашу мощную (человеческую) память и вспомнить нужную команду, которая позволяет скомпилировать и скомпоновать исходный текст GOODBYE.C. Вот подсказка:

```
gcc goodbye.c -o goodbye
```

Напечатайте эту команду в приглашении к вводу команды и смотрите на то, что происходит. Ну и...?

Ничего не происходит! Если вы сделали все должным образом, компилятор GCC просто создаст исполнимый программный файл. Вы увидите сообщение только в том случае, если допустили ошибку в программе.

Если вы действительно получили сообщение об ошибке, вероятнее всего, вы сделали опечатку или пропустили какой-нибудь крошечный символ: отсутствие “, или ;, или \, или), или (, или — вполне может привести к катастрофическому результату. Тщательно сравните (сверьте) исходный текст, приведенный в этой главе ранее, с тем, что вы ввели. С помощью редактора устраните ошибку, сохраните код на диске, а затем попытайтесь выполнить все снова.

⁵ Т.е. не нашел там серьезных ошибок. — Прим. ред.

Обратите внимание, что GCC, сообщая об ошибках, указывает номер строки, а иногда он может явно указать найденное им неправильно написанное слово (устранение ошибок в программах на C описывается в главе 2 “Ошибки в программах на C”).

Выполнение машинной программы

Если вы правильно указали команду компиляции, название (имя) машинной программы будет идентично первой части имени файла исходного текста. Так почему бы не выполнить эту программу?!

Для этого в Windows нужно напечатать команду

```
goodbye
```

В операционных системах семейства Unix необходимо указать путь к программе или ее местоположение перед названием (именем) программы. Напечатайте следующую команду:

```
./goodbye
```

Нажмите клавишу <Enter>, и программа выполнится, отображая на вашем экране следующий изумительный текст:

```
Goodbye, cruel world!
```

(До свидания, жестокий мир!)

Добро пожаловать в мир программирования на языке C!

(Более подробно выполнение программ описывается в приложении А “Прежде чем вы приступите к программированию”).

Сохраните исходный текст! Скомпилируйте и скомпонуйте программу! Выполните ее!

Требуются четыре шага, чтобы выполнить любую программу, исходный код которой написан на C. Это сохранение исходного текста, компиляция, компоновка и запуск (выполнение). Большинство пакетов для программирования на языке C автоматически выполняет шаг компоновки. Так что независимо от того, выполняется ли он вручную, он все равно выполняется.

Сохраните! (Команда Save.) Эта команда означает сохранение исходного текста. Вы создаете исходный текст в текстовом редакторе и сохраняете его как текстовый файл с расширением C (всего одна буква C).

Скомпилируйте и скомпонуйте! (Команды Compile и Link.) Компиляция — процесс преобразования команд, хранящихся в текстовом файле, в команды, которые может понять микропроцессор компьютера. На шаге компоновки команды преобразовываются в файл с программой. (Некоторые компиляторы могут выполнять этот шаг автоматически.)

Выполнить! (Команда Run.) Наконец, вы выполняете созданную вами же программу. Да, это — самая настоящая программа, как и любая другая на вашем жестком диске.

Завершив выполнение всех этих шагов, описанных в данной главе, вы создали программу GOODBYE. Теперь вы знаете, как создаются программы на С. На данном этапе труднее всего придумать, что написать в исходном файле, но эта задача по мере чтения данной книги будет становиться все проще. (А когда вы научитесь решать эту задачу, самым трудным будет получить правильно работающую безошибочную программу!)

Команды сохранения, компиляции и выполнения часто упоминаются в этой книге. Дело в том, что эти шаги являются более или менее механическими, к тому же важно понять, как применяется язык программирования. Именно об этом и пойдет речь в следующей главе.

Ошибки в программах на С

В этой главе...

- Перередактирование и перетранслирование
- Устранение ошибок
- Как разобраться в сообщениях об ошибках
- Как устранить отвратительные ошибки при компоновке

Путь первая успешная компиляция новой программы не создаст ложного представления о нормальном ходе рабочего дня программиста. На самом деле большая часть времени, отведенного на программирование, будет потрачена на устранение самых разнообразных ошибок, от простых опечаток до ошибок в логике. Все ошибки должны быть устранены. Однако они происходят настолько часто, что один знакомый гуру сказал, что программирование нужно было назвать отладкой.

В этой главе рассказывается об ошибках и их исправлении. Обратите внимание, что это вторая глава этой книги. Отсюда можно сделать вывод, что исправление ошибок в программировании намного важнее, чем вы, возможно, считали до этого.

Редактирование и перетрансляция — горькие слезы

Иногда люди допускают оговорки, заменяя по ошибке одно слово похожим, или вообще неправильно произносят тот или иной звук. Большое дело! Но только попробуйте напечатать `printf`, а не `printf`, и вся программа будет считаться ошибочной. Или, что еще хуже, только попробуйте пропустить какую-нибудь фигурную скобку. Отсутствие одной фигурной скобки может привести к тому, что весь экран будет заполнен непонятными сообщениями об ошибках.

Но ошибки — не позор. Не бойтесь их! Они обычны в программировании. Ошибки случаются. Вам придется устранять их. Это делается так:

1. Заново отредактируйте исходный текст и сохраните исправленный файл на диске.
2. Перетранслируйте исходный текст.
3. Выполните полученную программу.

Ошибки все еще могут остаться в программе. Может случиться даже так, что вы никогда не сможете добраться к шагу 3! Вот что нужно помнить.

- ✓ Ошибки случаются.
- ✓ Цикл разработки программы на языке С, приведенный в главе 1 “Основы языка С”, содержит шаги 4 и 6. Обойти их не удастся!

Очередное редактирование файла исходного текста

К счастью, исходный текст не нужно вырезать на камне или на кремниевой пластине. Его можно изменить. Иногда изменения (замены) необходимы для устранения ошибок и ляпов. Иногда же нужно изменить программу только для того, чтобы добавить новую возможность или изменить сообщение или подсказку. Для этого нужно снова отредактировать файл исходного текста.

Например, программа GOODBYE из главы 1 “Основы языка C” отображает на экране следующее сообщение:

Goodbye, cruel world!

(До свидания, жестокий мир!)

Эту программу легко изменить, чтобы отобразить любое нужное сообщение. Для этого с помощью редактора измените (замените) файл исходного текста, заменив первоначальное сообщение новым, более содержательным сообщением. Выполните следующие действия:

1. С помощью текстового редактора отредактируйте исходный текст GOODBYE.C.

2. Отредактируйте строку 5, в которой написано:

```
printf("Goodbye, cruel world!\n");
```

```
(printf ("До свидания, жестокий мир!\n"));
```

3. Замените текст Goodbye, cruel world! (До свидания, жестокий мир!) на Farewell, you ugly toad! (Прощайте, Вы уродливая жаба!).

```
printf("Farewell, you ugly toad!\n");
```

```
(printf ("Прощайте, Вы уродливая жаба!\n"));
```

Измените (замените) только текст между двойными кавычками. Это — информация, которая отображается на экране. Все остальное не трогайте!

4. Дважды перепроверьте вашу работу.

5. Сохраните файл на диске.

Теперь можно сохранить файл. Сохраняемый файл станет новым файлом исходного текста GOODBYE.C.

Далее можно перетранслировать исходный текст, об этом рассказывается в следующем разделе.



- ✓ “Заново отредактируйте файл исходного текста” означает, что с помощью текстового редактора нужно изменить исходный текст, т.е. текстовый файл, который содержит команды на языке C.
- ✓ Файл исходного текста редактируется для того, чтобы исправить ошибку, замеченную компилятором или компоновщиком, либо изменить программу. Это приходится делать много раз.
- ✓ Если редактор вызывается из командной строки, для повторного вызова предыдущих команд в некоторых средах в приглашении к вводу команды можно использовать клавишу со стрелкой “вверх”. Для этого несколько раз нажмите клавишу со стрелкой “вверх”, пока в подсказке не появится ранее введенная команда редактирования файла исходного текста GOODBYE.C.

Перетрансляция

В результате перетрансляции программа создается заново — ведь вы повторно выполняете шаги, которые уже выполнялись для создания программы. Это абсолютно необходимо после изменения или замены исходного текста, например, после замены, сделанной в предыду-

щем разделе. Поскольку исходный текст изменился, его нужно подать на вход компилятору снова, чтобы компилятор сгенерировал новую, лучшую (возможно даже безошибочную) программу.

Чтобы перетранслировать новый исходный текст GOODBYE.C, используйте ваш компилятор так, как указано в приложении А “Прежде чем вы приступите к программированию”. Возможно, достаточно будет ввести команду

```
gcc goodbye.c -o goodbye
```

Нажмите клавишу <Enter> и молитесь Богу, чтобы не появились сообщения об ошибках. Если сообщений не появилось, значит, была создана новая программа.

Теперь выполните программу! Напечатайте в строке с подсказкой надлежащую команду — goodbye или ./goodbye, чтобы увидеть новый, ошеломляющий результат.

Кто бы мог подумать, что на экране компьютера можно столь просто отобразить такое сногшибательное сообщение?



- ✓ Заново отредактировав файл исходного текста, его нужно перетранслировать, чтобы обновить файл программы. Именно так устраняются ошибки или изменяется программа.
- ✓ Если вы программируете в интегрированной среде разработки (IDE, Integrated Development Environment — Интегрированная среда разработки) вроде Dev-C++ или Microsoft Visual C++, для создания новой программы после изменения исходного текста можно воспользоваться командой Rebuild или Rebuild All.
- ✓ Если после перетранслирования обнаружатся ошибки, отредактированный ранее исходный текст нужно отредактировать заново и затем перетранслировать снова. (Ого, сколько “перередактирований” и “перетрансляций”!)

Торе от ошибок

Ошибки неизбежны. Даже лучшие из программистов получают сообщения об ошибках независимо от того, являются ли они невинными винтиками в Microsoft, пишущими код, или гордыми, независимыми программистами для Linux, — они все получают сообщения об ошибках. Каждый день.

Ошибки не должны вас смущать. Считайте их инструментами обучения или нежными напоминаниями. Просто компилятор очень точно определяет причину ошибки и место, где он ее обнаружил. Это полная противоположность злому педантичному учителю математики (кошмар!), который рядом с вашими вычислениями записал бы только “НЕПРАВИЛЬНО!”, независимо от того, насколько несущественна ваша ошибка. Да, компьютеры могут прощать — и это может даже научить вас кое-чему.

Ужас! Ошибка! Но не спешите стреляться....

Ниже приведена новая программа, ERROR.C. Обратите внимание на оптимизм в ее названии (имя ERROR означает ОШИБКА). Это испорченная программа на C, она содержит ошибку (притом нарочно сделанную ошибку):

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

```
int main() // главная функция
{
    printf("This program will err.\n") // программа с ошибкой
    return(0);
}
```

Напечатайте исходный текст точно так, как он приведен здесь¹. Не используйте исходный текст GOODBYE.C в качестве основы; начните с чистого окна редактора.

Введя исходный текст, сохраните его на диске под именем ERROR.C. Скомпилируйте его, а затем....

К сожалению, компилятор в этой программе обнаружит ошибку. В следующем разделе будет выяснена причина.

- ✓ Будьте внимательны при вводе программ! Каждый символ, будь то самая незаметная безобидная фигурная или круглая скобка, важен для языка C!
- ✓ Ниже приведена подсказка (обычная команда GCC), с помощью которой можно скомпилировать представленный выше исходный текст:

```
gcc error.c -o error
```

Это — последняя подсказка по компиляции программы в этой книге!

Поиск причины ошибки

В программе ERROR.C была обнаружена ошибка! Какой удар!

Но не тревожьтесь — это ожидалось. (На самом деле, возможно, вы уже встречались с этим типом ошибок.) Вот как может выглядеть это жестокое сообщение:

```
error.c: In function 'main':  
error.c:6: parse error before "return"
```

(Ошибка.c: В функции 'главная':

Ошибка.c:6: синтаксическая ошибка перед "возвратить")

Как грубо! Это вам не мягкий намек на то, что вы могли допустить незначительную неточность. Однако в сообщениях об ошибках, хотя и не учитывается индивидуальный подход, содержится важная информация.

Вот что главное: хотя сообщения об ошибках могут показаться загадочными, они информативны. Независимо от того, каким компилятором вы пользуетесь, вы должны научиться находить следующую информацию:

- ✓ Имя файла исходного текста, в котором была обнаружена ошибка. (В данном случае error.c.)
- ✓ Строка, в которой была обнаружена ошибка. (В данном случае строка 6.) Хотя на самом деле ошибка может быть и в другой строке — в этом вопросе компьютерам нельзя слишком доверять.
- ✓ Тип ошибки. (В данном случае ошибка при синтаксическом анализе (т.е. синтаксическая ошибка) или подобная ошибка.)
- ✓ Местоположение ошибки. (В данном случае перед словом return (возвратить).)

Несмотря на всю эту информацию, все еще может быть неясно, что именно неправильно, но компилятор дает вам много подсказок. Наиболее важная из них — номер строки: ошибка находится в строке 6.

¹ Комментарии (они начинаются с двух косых и расположены в правой части листинга) добавлены при переводе. В данном случае их набирать не нужно. — Прим. ред.

На самом деле ошибка в строке 5, но язык программирования C гибок, и компилятор не обнаруживает, что что-то пропущено, пока не дойдет до строки 6. (Вы можете воспользоваться этой гибкостью и исправление внести в строке 6 — компилятор допускает это.)

Тип ошибки также важен. Синтаксический анализ, или синтаксический разбор позволяет обнаружить синтаксическую ошибку, т.е. ошибку, состоящую в пропуске какого-нибудь знака пунктуации языка C, поэтому две вещи, которые, в соответствии с синтаксисом языка C, не могут следовать одна за другой, встретились вместе. В данном случае все дело в отсутствии символа “точка с запятой” в конце строки 5.

Решение? Вы должны отредактировать заново файл исходного текста и исправить то, что является неправильным. В данном случае нужно заново отредактировать ERROR.C и добавить точку с запятой в конце строки 5. Даже если будете очень внимательно рассматривать строку 6, все равно вы не увидите там ничего неправильного. Но если бы вы посмотрели немного назад, вы бы увидели ошибку и исправили бы ее. Если не можете найти ошибку в указанной компилятором строке, вспомните о Синдроме Отсутствующей Точки с запятой!



- ✓ Ошибки — это еще не конец света! Каждый получает сообщения о них.
- ✓ Синтаксис определяет способ соединения элементов языка. Иными словами, синтаксис определяет порядок соединения элементов языка (немного мудрено).
- ✓ В некоторых версиях GCC слово “return” заключается в двойные кавычки, а в некоторых — в одинарные.
- ✓ Отсутствие точки с запятой — один из самых частых типов ошибок в программах на языке C. О точках с запятой и их роли вы узнаете в следующей главе.
- ✓ Номер строки в сообщении об ошибке относится к строке в текстовом файле исходного текста. Именно поэтому почти во всех текстовых редакторах номера строк отображаются сверху или внизу экрана или окна редактирования.
- ✓ Номер строки может быть указан неточно. В случае отсутствия точки с запятой, ошибка может быть обнаружена лишь в следующей строке. Это утверждение верно и для других типов ошибок в программах на C. Но, во всяком случае, ошибка где-то близко, и, что еще лучше, динамик при этом не гудит и экран не заполняется сообщением “ОШИБКИ В ИЗОБИЛИИ, ВЫ ПОЛНЫЙ ИДИОТ” даже в присутствии вашего шефа.
- ✓ Лучше всего исправить первую указанную ошибку, а затем программу перетранслировать. Часто случается, что первая ошибка как раз является единственной настоящей ошибкой, а компилятор указывает еще и другие, потому что он просто путается.
- ✓ Конечно, вы можете подумать: “Хорошо, компьютер, ты хитер и остроумен, ты знаешь, что неправильно — так исправь это!” Однако компьютеры подобных умозаключений не понимают. Причиной этого зла является принцип “Делай то, что я подразумеваю”. Но на самом деле компьютеры не могут читать мысли, так что вы должны быть точны. Компьютеры чемпионы лишь по скорости поиска ошибок, однако исправить их они не могут (и это относится почти ко всем снобам).

Исправление ошибочной программы

Чтобы снова получить правильную программу, нужно ее исправить. Для этого нужно отредактировать файл исходного текста, внеся необходимое исправление, сохранить файл исходного текста на диске, а затем перетранслировать его.

Чтобы исправить программу ERROR.C, достаточно добавить точку с запятой. Отредактируйте строку 5 — добавьте точку с запятой в конце строки. А также исправьте предложение, отображаемое на экране, следующим образом²:

```
// в программе больше нет ошибки  
printf("This program will no longer err.\n");
```

Кроме измененной строки 5, все остальное в программе остается нетронутым.

Снова сохраните ERROR.C на диске. Перетранслируйте программу, и затем выполните ее:
This program will no longer err.

(Эта программа больше не будет ошибаться.)

Действительно, не ошибается!

- ✓ Я не могу всегда говорить вам, где в программе нужно исправить ошибку. ERROR.C — единственная программа, в которой ошибка была сделана нарочно. Когда вы получите сообщение об ошибке, вы должны понять его смысл, чтобы увидеть, где в исходном тексте допущена ошибка. Затем перепроверьте исходный текст и сверьте его с тем, который приведен в данной книге. Таким способом вы найдете то, что неправильно. Но когда вы будете писать программы самостоятельно, у вас будет только сообщение об ошибках. Оно поможет вам выследить ваши собственные ошибки.
- ✓ Удалите две буквы R из ERRORS (ОШИБКИ), и у вас получится Eros (Эрос), греческий бог любви. Римским богом любви был Купидон (Cupid). Замените C в Cupid (Купидон) на St, и у вас получится Stupid (Глупый). Глупые ошибки — как прекрасно! Мораль: незначительные изменения могут привести к серьезным последствиям.



Предупреждения и ошибки — не забивайте этим вашу голову

При программировании на C различают две степени ошибок: предупреждения и ошибки. Некоторые компиляторы называют ошибки критическими (critical errors), другие — неустраняемыми, фатальными ошибками (fatal errors).

Ошибки-предупреждения компилятор трактует примерно так: "О-о-о, это не выглядит вкусным, но я подам его вам так или иначе". Возможно, ваша программа выполнится, но не сможет сделать то, что вы хотите. Или, может статься так, что ошибка только подразнит компилятор. В большинстве компиляторов C многие назойливые предупреждающие сообщения об ошибках можно выключить.

Критическая ошибка означает: "Дорогой Господин, Вы пробовали сделать кое-что столь преступное, что из нравственных соображений я не могу закончить эту программу". Ладно, возможно все не так серьезно. Но компилятор не может выполнить свою задачу, потому что он совсем не понимает ваши команды.

А теперь попробуйте исправить эту ошибку!

Давайте не будем ограничиваться исправлением только одной ошибки в файле ERROR.C. Не закрывайте окно и не удаляйте проект. (Если вы это уже сделали, загрузите в редактор файл ERROR.C и приготовьтесь редактировать его заново.)

² Первая строка, как вы, вероятно, уже догадались, — комментарий (притом добавленный при переводе для удобства русскоязычных читателей). В данном случае набирать его не нужно. — Прим. ред.

Измените строку 6 в файле исходного текста ERROR.C следующим образом:

```
retrun(0);
```

В случае если вы не видите, что это новая ошибка, замечу, что слово `return` искорежено, оно заменено словом `retrun`, которое отличается от `return` перестановкой второй буквы `r` и `u`. А вот ноль в круглых скобках и точка с запятой остались на своих местах.

Однако язык C устроен так, что компилятор предполагает, что `retrun` — нечто вполне серьезное, а не опечатка. Компилятор бездумно проглотил его как команду. Но компоновщик сходит с ума из-за этого. Дело в том, что именно компоновщик связывает программные файлы. Он ищет слово `retrun` во всех своих библиотеках, но не находит его там. И тогда, подобно любому измотанному библиотекарю, компоновщик³ ругается — он выдаст сообщение об ошибках.

Сохраните измененный файл ERROR.C на диске. Затем перетранслируйте его. Приготовьтесь прочитать несколько строк сообщений об ошибках.

```
temporary_filename.o: In function 'main':  
temporary_filename.o: undefined reference to 'retrun'
```

(temporary_filename.o: В функции 'главная':

temporary_filename.o: неопределенная ссылка на 'retrun')

Впрочем, сообщение может выглядеть иначе, примерно следующим образом:

```
temporary_filename.o(и т.п.):error.c: undefined reference to 'retrun'
```

(temporary_filename.o (и тому подобное):error.c: неопределенная ссылка на 'retrun')

Теперь труднее сказать, где была допущена ошибка; в отличие от ошибок, обнаруживаемых компилятором, определить место ошибки, обнаруженной компоновщиком, гораздо труднее. В данном случае компоновщик объясняет, что ошибка состоит в ссылке на слово `retrun`. Так что вместо того, чтобы просто найти строку с указанным номером, программисту приходится искать ошибочный текст.

Чтобы исправить ошибку, заново отредактируйте исходный текст и замените слово `retrun` тем, которое было, чтобы программа снова была правильной. Сохраните файл. Перетранслируйте. Теперь компоновщик снова должен быть доволен!

- ✓ Как уже упоминалось ранее, компилятор GCC и компилирует программу, и компо-
нует ее.
- ✓ Если компоновщик запускается как отдельная программа, он, очевидно, генерирует
свои собственные сообщения об ошибках.
- ✓ Компилятором создается временный файл — файл объектного кода, имя которого
заканчивается на `.O`. Имя этого временного файла вы и видите в листинге сообще-
ний об ошибках. Этот файл объектного кода удаляется компилятором GCC.
- ✓ Компоновщик связывает различные части программы. Если он находит нечто, что
не может распознать, например `retrun`, он думает, что, возможно, это находится в
другой части программы. Так проскальзывают ошибки. Но впоследствии, когда
компоновщик пробует найти отсутствующее слово, он понимает, что это ошибка,
и генерирует сообщение.

³ Было время, когда термины *библиотекарь* и *компоновщик* считались синонимами в программировании. — Прим. ред.



Все об ошибках!

Общепринятая аксиома программирования гласит, что вы не можете создать компьютерную программу до тех пор, пока не исправите все ошибки в ней. Ошибки повсюду, и исправление их может занять годы, пока, наконец, не будет написано хорошее программное обеспечение.

Ошибки, обнаруживаемые компилятором. Самый распространенный тип ошибок. Первоначально ошибки этого типа обнаруживаются компилятором, когда он пробует прочитать текст, записанный программистом, и перевести его в команды, которые может понять компьютер. Эти ошибки — дружелюбные, они очевидны и сопровождаются указанием номеров строк и всеми прочими детальными описаниями. Эти ошибки будут замечены еще до создания компьютерной программы.

Ошибки, обнаруживаемые компоновщиком. Чаще всего они возникают из-за орфографических ошибок при записи команд. Если вы используете дополнительные возможности программирования на С, например, работаете с несколькими исходными файлами или модулями, создавая большую программу, ошибки, обнаруживаемые компоновщиком, могут возникнуть из-за отсутствия нужных модулей. Кроме того, если компоновщику требуется некоторый "библиотечный" файл, а он его не может найти, отображаются сообщения об ошибках еще одного типа. Хотя части программы скомпилированы без синтаксических ошибок, оставшиеся ошибки не позволяют скомпоновать программу в единое целое.

Ошибки, обнаруживаемые во время выполнения программы. Генерируются программой во время ее выполнения. Эти ошибки — не мелкие дефекты; они полностью незаметны для компилятора и компоновщика, но именно из-за них программа не делает то, для чего вы ее создали. (Это часто случается при программировании на С.) Самая частая ошибка, обнаруживаемая во время выполнения программы, — нулевое (пустое) значение указателя. Вы узнаете об этом позже. Машинная программа создается, но часто завершается операционной системой при запуске.

Ошибки-дефекты. Есть еще один тип ошибок. Компилятор старательно создает написанную вами программу, но делает ли созданная программа то, что вы задумали, — выясняется в ходе ее испытания. Если она не делает то, что вы задумали, придется потрудиться над исходным текстом еще некоторое время. Ошибки-дефекты могут проявляться самым неожиданным образом — от замедленной работы программы вплоть до того, что программа работает совсем не так, как задумано, или не работает вообще. Именно ошибки этого типа труднее всего выловить и именно они обычно доставляют больше всего хлопот. Машинная программа создается и выполняется, но ведет себя совсем не так, как вы хотите.

Формальное знакомство с языком C

В этой главе...

- Знакомство с языком C
- Разделение исходного текста
- Следование ПРАВИЛАМ
- Использование \n
- Разбиение строк с помощью \

Для новичка любой новый язык выглядит сверхъестественно. В родном языке вы привыкли к определенной интонации и способам построения слов. А все символы соединяются знакомым вам способом. Иностранные языки имеют необычные символы вроде ç, ü и Ø, а также комбинации символов, которые выглядят странными для тех, кто привык говорить по-английски: Gwynedd (Гуинет), Zgierz, Qom, и Idaho (Айдахо).

К этим новым странным словам приходится привыкать. Нужно знать, что есть что. Нет смысла вслепую печатать программу на C, если вы не имеете представления, что означают слова в ней. Именно поэтому в данной главе разъясняется смысл слов, которые встретились вам в предыдущих программах. После чтения двух глав и компиляции двух различных программ на C и исправления ошибок, в этой главе, наконец, вы начнете знакомиться с правилами языка C.

Большой рисунок

На рис. 3.1 выделены основные элементы исходного текста программы GOODBYE.C, которую я использовал в качестве примера в главе 1 “Основы языка C”.

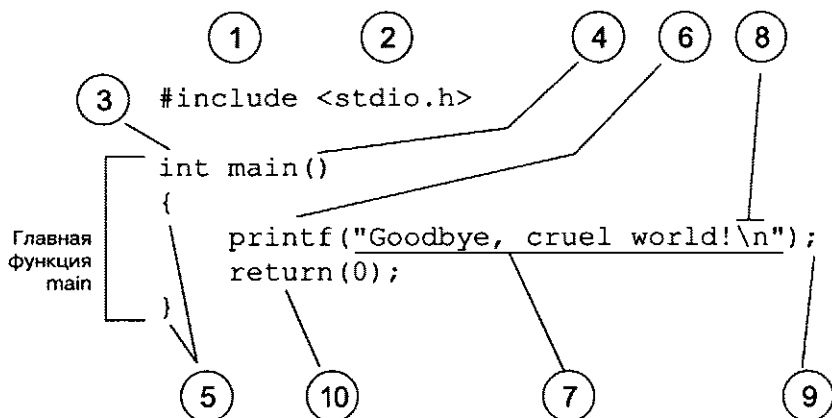


Рис. 3.1. Программа GOODBYE.C и ее части

Каждая программа должна иметь точку входа. Когда вы выполняете программу, операционная система (ОС) запускает ее. Иными словами, ОС указывает, что микропроцессор должен начать выполнение программы. Затем микропроцессор начинает выполнение программы с определенной отправной точки — эта точка и называется точкой входа в программу.

Во всех программах на С отправной точкой служит главная функция — функция `main()`. Каждая программа на С имеет ее; например, программы `GOODBYE.C`, `ERROR.C` (и все другие программы на С) содержат эту функцию. Главная функция — функция `main()` — механизм, который заставляет работать всю программу. Главная функция — функция `main()` — представляет собой скелет, или каркас, на который навешивается остальная часть программы.

- ✓ `main()` — название (имя), которое дается первой (главной, начальной, т.е. той, с которой начинается выполнение) функции в каждой программе на С. Программа на С может иметь и другие функции, но первой начинает выполняться главная функция — `main()`.
- ✓ В языке С после имени функции следуют круглые скобки. Круглые скобки могут быть либо пусты, либо они могут содержать некоторую информацию — в зависимости от конкретной функции.
- ✓ Обычно в книгах (в частности, в этой) после имен функций, записанных на языке С, часто ставят круглые скобки. Например, вот так: `main()`.
- ✓ Функция — это машина, которая умеет выполнять набор некоторых команд, каждая из которых что-то делает. Программы на С могут содержать много функций, хотя первой функцией по порядку выполнения в программе С должна быть главная функция — функция `main()`. Это обязательное требование.

Функция. Привыкните к этому слову.

Составные части программы на языке С

Обратите внимание на некоторые интересные части программы С, показанной на рис. 3.1:

1. Со слова **#include** начинается *директива препроцессора*; звучит внушительно, однако этот термин не может рассматриваться как абсолютно правильный, во всяком случае вы не обязаны его запоминать. Это слово предписывает компилятору “включить” текст с другого файла, т.е. вставить его прямо в ваш исходный текст. Благодаря этому можно избежать большого количества мелких раздражающих ошибок, которые произошли бы в противном случае в результате ввода текста.
2. `<stdio.h>` — имя файла в угловых скобках (в языке С приходится использовать все виды скобок). Инstrukция `#include <stdio.h>` заставляет компилятор взять текст из файла `STDIO.H` и вставить его в исходный текст до компиляции исходного текста. Сам файл `STDIO.H` содержит информацию о стандартных функциях ввода-вывода (STanDard Input/Output), требуемых большинством программ на С. Буква Н означает “header” (“заголовок”). Подробнее о заголовочных файлах будет рассказано в главе 23 “То, что пишется в начале программы”.
3. Слова `int main` служат двум целям. Во-первых, `int` указывает, что главная функция является целочисленной функцией; это означает, что после выполнения главная функция `main()` должна вернуть целочисленное значение. Во-вторых, в этой строке начинается определение главной функции `main`, с которой начинается выполнение программы.

Подробнее о значениях, возвращаемых функциями, читайте в главе 22 “Как на самом деле функционируют функции”.

4. За именем функции следует пара пустых круглых скобок. Иногда в этих круглых скобках могут быть заключены некоторые элементы программы, о чем рассказывается в главе 22 “Как на самом деле функционируют функции”.
5. Все содержимое функции на C заключается в фигурные скобки. Поэтому сначала написано имя функции (`main` в выноске 3), а затем ее содержимое — его можно рассматривать как машину, которая выполняет задачу функции (содержимое заключено в фигурные скобки).
6. Слово `printf` — название (имя) функции на языке C, так что в соответствии с нашим соглашением, упоминая эту функцию, мы будем записывать ее как `printf()`. Ее задача — отображать информацию на экране. (Поскольку принтеры предшествовали компьютерным мониторам, команды, отображающие информацию на экране, назывались командами *печати*. Добавление *f* означает “форматированный” (“formatted”), об этом вы узнаете подробнее в следующих главах.)
7. Как и после всех имен функций в языке C, в `printf()` после имени функции идет пара круглых скобок. В круглые скобки заключается текст, или “строка” символов. Все, что расположено между символами двойных кавычек (“”), является частью текстовой строки, выводимой функцией `printf`.
8. Интересная часть текстовой строки — `\n`. Это символ наклонной черты влево и строчная буква *n*. Так в C изображается символ *новой строки* (newline), этот символ заносится в файл тогда, когда вы нажимаете клавишу Enter. Вы узнаете больше об этом и других сверхъестественных комбинациях символов с наклонной чертой влево в главе 7 “*A + B = C*”.
9. Строка (инструкция) `printf` заканчивается точкой с запятой. Точка с запятой — символ пунктуации в языке C — подобна запятой в английском языке¹. Точка с запятой говорит компилятору языка C, где заканчивается один оператор и начинается следующий. Обратите внимание, что в C все инструкции заканчиваются точкой с запятой, даже если в программе или функции есть только одна инструкция.
10. Вторая инструкция в `GOODBYE.C` — команда возврата. Эта команда посылает значение 0 (ноль) на операционной системе, когда главная функция (функция `main()`) завершается. Каждая функция должна возвращать значение (почему, вы узнаете из главы 22 “Как на самом деле функционируют функции”). Обратите внимание, что хотя эта команда — последняя в программе, данный оператор заканчивается точкой с запятой.

- ✓ Текст в программе называется строкой. Например, “la-de-da” — строка текста. Строка заключается в двойные кавычки.
- ✓ Функция на языке C начинается с типа функции, например, с типа `int`, а затем следует имя функции и круглые скобки, как вы уже это видели в главной функции `main()`. Затем в фигурных скобках { } следует тело функции. Все между { и } является частью функции.
- ✓ Язык C состоит из ключевых слов, которые используются в инструкциях. Инструкции заканчиваются точкой с запятой, так же, как в конце английских предложений ставятся точки². (Но не утруждайте себя запоминанием этого правила в данный момент.)

¹ Да и в русском. — Прим. ред.

² И русских. — Прим. ред.

Лексика языка C: ключевые слова

Язык C действительно довольно краток. Язык C имеет только 32 ключевых слова. Вот если бы французский язык был таким легким! В табл. 3.1 приведены ключевые слова, которые составляют лексику языка C.

Таблица 3.1. Ключевые слова языка C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

Неплохо, да? Но это не все слова, которые вы используете при записи программ на C. Другие слова или команды называют функциями. Среди них есть такие драгоценные названия, как `printf()` и еще несколько десятков других общих функций, которые вместе с основными ключевыми словами языка C помогают в создании программ.

Кроме ключевых слов, языки программирования (подобно человеческим языкам) имеют грамматику, т.е. правила правильного соединения слов, чтобы можно было ясно выразить (передать) мысли (идеи). Это то, что полностью игнорируют современные юристы. (Вы пробовали обнаружить смысл в их законах?)

В дополнение к грамматике, в языках есть правила, исключения, диакритические знаки, а также всех сортов забавы и приколы. Языки программирования подобны разговорным языкам в том, что они имеют различные аспекты и много правил.

- ✓ Ключевые слова могут также называться зарезервированными словами.
- ✓ Обратите внимание, что все ключевые слова набираются на нижнем регистре. Это непреложная истина для C: ключевые слова, так же как и названия (имена) функций, набираются только на нижнем регистре. Язык C чувствителен к регистру, так что для него есть различие между `return`, `Return` и `RETURN`.
- ✓ Вы ни в коем случае не обязаны запоминать эти 32 ключевых слова.
- ✓ На самом деле из этих 32 ключевых слов только половина используется систематически.
- ✓ Некоторые ключевые слова — настоящие слова! Другие — сокращения или комбинации двух (или большего количества) слов. А есть просто криптограммы имен подруг программистов.
- ✓ Каждое ключевое слово имеет свое собственное предназначение. Нельзя, например, использовать ключевое слово `else` где попало; его можно использовать только в определенном контексте.
- ✓ Для вызова некоторых функций, например `printf()`, требуется не просто пара круглых скобок, а еще и большое количество всякой всячины в круглых скобках. (Не беспокойтесь обо всех деталях этой инструкции прямо сейчас; достаточно только подумать: “Да, `printf()` действительно требует большого количества всякой всячины”.)



- ✓ Между прочим, именно по той причине, что `printf()` — функция C, а не ключевое слово, как раз и требуется в начале программы строка `#include <stdio.h>`. Файл `STDIO.H` содержит команды, которые указывают компилятору, чем именно является `printf()` и что она делает. Если выбросить строку `#include <stdio.h>`, компилятор с испугу сгенерирует ошибку типа “я не знаю, что сделать с этой штукой `printf()`”.



Эти безумные ключевые слова!

Ключевые слова стоят упоминания именно потому, что их использование ограничено или зарезервировано. Например, нельзя придумать какую-нибудь функцию и назвать ее `short` (короткий). Дело в том, что `short` — ключевое (зарезервированное) слово, имеющее только одно определенное предназначение в ядре языка C. Именно в этом смысле ключевые слова являются специальными.

В дополнение к этим 32 ключевым словам, приведенным в табл. 3.1, есть еще два (теперь уже устаревших и потому обесценившихся) ключевых слова языка C: `fortran` и `entry`.

Когда-то в C эти слова были ключевыми, но больше они таковыми не считаются. Однако я не рекомендую бы использовать их в программах. (Так что они действительно “обесценились”.)

Кроме того, в языке C++ есть зарезервированные слова. Если Вы планируете изучать C++, включать эти слова в ваш словарь “неиспользуемых”, резервированных слов языка C:

<code>asm</code>	<code>false</code>	<code>private</code>	<code>throw</code>
<code>bool</code>	<code>friend</code>	<code>protected</code>	<code>true</code>
<code>catch</code>	<code>inline</code>	<code>public</code>	<code>try</code>
<code>class</code>	<code>mutable</code>	<code>reinterpret_cast</code>	<code>typeid</code>
<code>const_cast</code>	<code>namespace</code>	<code>static_cast</code>	<code>using</code>
<code>delete</code>	<code>new</code>	<code>template</code>	<code>virtual</code>
<code>dynamic_cast</code>	<code>operator</code>	<code>this</code>	

Изучите эти слова теперь и постарайтесь избегать их, поскольку если вы привыкните использовать какое-нибудь слово (например, `new` или `friend`), то столкнетесь с неприятностью позже, когда будете работать с языком C++.

Другие компоненты языка C

Язык C имеет много других особенностей, и потому выглядит довольно причудливо для программиста-новичка. Сейчас вы находитесь между невежеством и знанием — в свое время вы научитесь всем тонкостям, а пока не закидывайтесь на том, чего не знаете. Вместо этого обратите внимание на следующее:

- ✓ Язык C использует слова (ключевые слова, имена функций и т.д.) в качестве основных элементов.
- ✓ Вместе со словами используются и символы. Иногда эти символы называют *операторами*, а иногда их называют еще иначе. Например, в C знак “плюс” (+) обозначает сложение.
- ✓ Слова имеют опции и подчиняются правилам их употребления. Эти правила перечислены в справочнике по C, который поставляется вместе с компилятором. Вы не обязаны помнить все правила, хотя некоторые из них вы впоследствии будете применять автоматически.

- ✓ Круглые скобки используются для группировки некоторых важных элементов, требуемых словами языка С.
Слова соединяют для того, чтобы создать инструкции, которые подобны предложениям английского языка. Все инструкции заканчиваются точкой с запятой.
- ✓ Фигурные скобки используются для группирования частей программы. Некоторые слова используют фигурные скобки, чтобы группировать то, что им принадлежит, и все функции программы для группирования своих инструкций используют фигурные скобки. На рис. 3.1 и во всех программах на С в первых двух главах, например, фигурные скобки использовались для того, чтобы поместить в них содержимое главной функции `main()`.
- ✓ Весь этот материал (а также дополнительный материал, который я не осмеливаюсь обсуждать в этой главе) составляет синтаксис языка С. *Синтаксис* — это способ построения предложений языка.

Викторина!

1. Основную (ту, с которой начинается выполнение программы) функцию в каждой программе на языке С называют
 - а) `numero_uno()`;
 - б) главной функцией `main()`;
 - в) `primus()`;
 - г) ядром `core()`.
2. Ключевые слова языка С
 - а) являются “словами” языка С;
 - б) скрепляются строками и воском;
 - в) произносятся с особым почтением Гуру Языка С и только при свечах;
 - г) столь же многочисленны, как звезды, и почти так же недостижимы.
3. В дополнение к ключевым словам в языке С используются
 - а) функции, типа `printf()`;
 - б) операторы, такие как `+`, `-` и другие сверхъестественные вещи;
 - в) фигурные и угловые скобки, а также все мыслимые и немыслимые виды скобок;
 - г) вероятно, все перечисленное выше.
4. Функции требуют круглых? скобок потому что:
 - а) они говорят шепотом;
 - б) круглые скобки обогрывают функцию;
 - в) в круглых скобках можно объединить различные вещи, требуемые для функции;
 - г) А что такое функция?
5. Верный признак любой программы на С — фигурные скобки. Используя ваши знания о С, вставьте фигурные скобки там, где они нужны в следующей программе:


```
int main() // главная функция
{
    printf("Goodbye, cruel world!\n"); // До свидания, жестокий мир!
    return(0);
}
```

Ответы см. на странице 516³.

Полезная программа RULES (ПРАВИЛА)

Я просто не могу оставить эту главу без программы. Чтобы помочь вам запомнить наиболее часто нарушаемые правила языка С, я суммировал их в следующей программе. Она отображает несколько строк текста, напоминающих вам об основных правилах С, которые вы уже знаете:

```
#include <stdio.h>

int main() // главная функция
{
    // Фигурные скобки ходят парами!
    printf("Braces come in pairs!");

    // Комментарии ходят парами!
    // Все инструкции заканчиваются точкой с запятой!
    printf("Comments come in pairs!");
    printf("All statements end with a semicolon!");
    printf("Spaces are optional!"); // Пробелы не обязательны!
    // Должна быть основная функция!
    printf("Must have a main function!");
    // Почти все на С печатается на нижнем регистре
    // Этот язык учитывает регистр.
    printf("C is done mostly in lowercase.\n");
    printf("It's a case-sensitive language.");
    return(0);
}
```

Введите предыдущий исходный текст в вашем редакторе. Сохраните код на диске под именем RULES.C. Скомпилируйте и запустите программу.

Полученная программа будет называться RULES (ПРАВИЛА), и вы сможете выполнить ее в любой момент, когда вам нужно будет напомнить некоторое основное правило этикета в языке С.

- ✓ Эта программа в действительности почти не содержит элементов, отличных от приведенных в предыдущих главах. В ней просто больше вызовов функции `printf()`.
- ✓ Заключительный вызов функции `printf()` (в четвертой от конца строке) может казаться немного странным. Дело в том, что он разбит на две строки (такое хитрое изобретение объясняется в этой главе несколько позже).

³ Это авторский прикол. Автор, видимо, надеется, что вы прочитаете книгу до конца, ведь всего в оригинале 390 страниц. К разряду приколов нужно отнести, видимо, и эту викторину, ведь автор обещал, что их не будет в этой книге. Тем не менее, я уверен, размышление над вопросами не отнимет у вас много времени. Ну, а если для ответа на какой-либо вопрос понадобится более пяти минут, польза от викторины будет очевидна — вам придется повторить крепко-накрепко забытый материал. — Прим. ред.

Важность наличия \n

Вы обратили внимание на нечто странное в выводе программы RULES? Да, это напоминает уродливый комок текста:

```
Braces come in pairs!Comments come in pairs!All statements end with a
semicolon!Spaces are optional!Must have a main function!C is done mostly
in lowercase. It's a case-sensitive language.
```

(Фигурные скобки ходят парами! Комментарии ходят парами! Все инструкции заканчиваются точкой с запятой! Пробелы не обязательны! Должна быть основная функция! Почти все на С печатается на нижнем регистре. Этот язык учитывает регистр.)

Исходный текст вроде бы в порядке, но в выводе отсутствует символ, который получается при нажатии клавиши <Enter>, т.е. то, что называется символом новой строки (newline). Вы видели его прежде — этот необычный символ \n.

Этот символ в С означает переход на новую строку текста. Буква n означает *new* (новая) в *new line* (новая строка). (Правда, программисты записывают это словосочетание как одно слово: *newline*.)

Новую программу RULES.C нужно отредактировать, добавив символ \n в конце каждой строки, — это гарантирует, что каждая строка будет отображена на отдельной строке экрана. Благодаря этому добавлению читать вывод программы RULES будет гораздо проще.

Снова отредактируйте файл исходного текста RULES.C. Перед последней двойной кавычкой в каждой строке вызова `printf()` добавьте символ \n.



Если вы хорошо знакомы с поиском и заменой, ищите последовательность ") (кавычка-скобка) и заменяйте ее на "\n").

Сохраните файл на диске и перетранслируйте его. Вывод должен теперь читаться легче:

```
Braces come in pairs!
Comments come in pairs!
All statements end with a semicolon!
Spaces are optional!
Must have a main function!
C is done mostly in lowercase. It's a case-sensitive language.
```

(Фигурные скобки ходят парами!

Комментарии ходят парами!

Все инструкции заканчиваются точкой с запятой!

Пробелы не обязательны!

Должна быть основная функция!

Почти все на С печатается на нижнем регистре. Этот язык учитывает регистр.)

- ✓ В текстовой строке языка С используется символ \n, это выглядит так, как если бы была нажата клавиша <Enter>.
- ✓ В \n буква n всегда строчная. Программы на С набирать следует главным образом на нижнем регистре.
- ✓ В главе 24 “Глава о функции `printf()`” в табл. 24.1 перечислены и другие символы, подобные \n.

Разбить строки символом \ очень просто

Другая аномалия программы RULES — тот жуликоватый символ \ в четвертой от конца строке. Когда он используется в конце строки, единственный символ \ говорит компилятору, что остальная часть строки просто продолжена на следующей строке. Так, следующие две строки:

```
printf("C is done mostly in lowercase.\n\n");
    It's a case-sensitive language.");
```

(Почти все на С печатается на нижнем регистре.

Этот язык учитывает регистр.)

компилятором рассматриваются как одна строка, и на самом деле компилятор видит вот это:

```
printf("C is done mostly in lowercase. It's a case-sensitive language.\n\n");
```

(“Почти все на С печатается на нижнем регистре. Этот язык учитывает регистр.”);

По мере усложнения ваших программ такая уловка может понадобиться для экстра-длинных строк. Символ \ просто позволяет разбивать длинную строку на несколько строк исходного текста. Это может быть удобно.

В зависимости от вашего редактора и компилятора, результат разбиения строки на несколько строк исходного текста может быть не таким, как вы хотите. Например, вывод на экране может выглядеть следующим образом:

```
Braces come in pairs!
Comments come in pairs!
All statements end with a semicolon!
Spaces are optional!
Must have a main function!
C is done mostly in lowercase.           It's a case-
sensitive language.
```

(Фигурные скобки ходят парами!

Комментарии ходят парами!

Все инструкции заканчиваются точкой с запятой!

Пробелы не обязательны!

Должна быть основная функция!

Почти все на С печатается на нижнем регистре.

Этот язык

учитывает регистр.)

Это случается потому, что разбитая строка включает несколько позиций табуляции, чтобы выдержать отступы, — благодаря этому она выглядит симпатично в вашем исходном тексте, но когда программа выполняется, результат напоминает строки, из которых что-то вымарали. Решение состоит в том, чтобы просто отредактировать исходный текст, а именно удалить дополнительные позиции табуляции из строки текста.

Давайте теперь изменим четвертую и третью строки от конца исходного текста. Вот что имеем:

```
printf("C is done mostly in lowercase.\n\n");
    It's a case-sensitive language.");
```

(Почти все на С печатается на нижнем регистре.

Этот язык учитывает регистр.)

Заменяем эти строки следующими:

```
printf("C is done mostly in lowercase. \n\n");
    It's a case-sensitive language.");
```

(Почти все на С печатается на нижнем регистре.

Этот язык учитывает регистр.)

Обратите внимание на дополнительный пробел после точки в первой строке (перед наклонной чертой влево), который разделяет две строки текста, — иначе бы они слились друг с другом. Сохраните получившийся не очень красивый файл. Скомпилируйте и выполните. Вывод будет намного приятнее для глаз.



- ✓ Обратите внимание, что в некоторых других программах из этой книги тоже может использоваться \ для разбиения длинных строк.
- ✓ Не волнуйтесь по поводу использования \ для разбиения строк. Это — уловка, которую иногда вы встретите в программах некоторых программистов, но я предпочитаю прокручивать окно редактора и разбивать длинные строки с помощью редактора.
- ✓ Хотя разбитые строки обрабатываются как одна строка, для всех ошибок, которые встречаются на любой строке, даются номера их строк в файле исходного текста. Так, если бы точка с запятой отсутствовала в конце предпоследней строки в примере программы RULES.C, то компилятор пометил бы эту ошибку номером этой строки или следующей, а не предшествующей ей строки⁴.

⁴ Иными словами, ошибка была бы отнесена к строке с номером n или $n+1$, а не к строке с номером $n-1$ или n , если номер предпоследней строки обозначить через n . Все зависит, конечно, от компилятора, но вероятнее всего, ошибка была бы замечена в строке с номером n или $n+1$. — Прим. ред.

Что такое ввод-вывод?

В этой главе...

- Ввод с клавиатуры
- Функция `printf()`
- Создание форматированного вывода
- Функция `scanf()`

В компьютерах все связано с вводом и выводом (да, с тем самым вводом-выводом, о котором когда-то пели первопроходцы). Женщины тогда любили очень медленно танцевать. Возможно, плакать. Это все было так сентиментально. Обо всем этом современные горожане теперь читают в дешевых журналах.

Гм!

Ввод и вывод: вы вводите что-нибудь и видите реакцию, задаёте вопрос и получаете ответ, бросаете монеты в автомат и получаете газировку — так все и происходит. Это как раз то, о чем я рассказываю в главе 3 “Формальное знакомство с языком C”: задача программиста состоит в том, чтобы написать программу, которая будет что-то делать. На данном этапе изучения лучше рассматривать тривиальные вещи. Однако вскоре вы начнете писать программы, которые действительно делают полезные вещи.

Представьтесь компьютеру

Познакомиться с вводом и выводом (имеются в виду старые средства ввода-вывода) можно с помощью программы WHORU.C. WHORU — это я так сократил фразу “who are you” (“кто вы?”), выкинув из нее несколько букв. Но не называйте эту программу “horror-you” (страшилка), все-таки это семейная книга. Впрочем, название программы можно истолковать и совсем иначе, но это уж точно не для семейной книги...

Эта программа вводит ваше имя с клавиатуры, а затем заставляет компьютер отобразить дружеское приветствие, вставив в него ваше имя.

```
#include <stdio.h>
int main() // главная функция
{
    char me[20];

    printf("What is your name?"); // Как вас зовут?
    scanf("%s",&me);
    // Приятно познакомиться
    printf("Darn glad to meet you, %s!\n",me);
    return(0);
}
```

Наберите эту исходную программу в вашем редакторе. Проверьте все несколько раз. Пока не беспокойтесь о деталях. Набирая, можно напевать, если вам это нравится.

Сохраните файл на диске. Назовите его WHORU.C.

Пока не компилируйте эту программу. Отложим это до следующего раздела.



- ✓ `char me[20];` — это *объявление переменной*. Оно отводит память для информации, которую вы вводите (“ввод” во вводе-выводе). Более подробно о переменных будет рассказано в главе 8 “Переменные в языке C”.
- ✓ Новой функцией здесь является `scanf()`. Она используется для считывания информации, введенной с клавиатуры, и сохранения ее в памяти компьютера.
- ✓ Левая скобка — символ `(`. Правая скобка — символ `)`. Скобки — именно так для краткости называют круглые скобки.

Компиляция файла WHORU.C

Откомпилируйте исходную программу WHORU.C. Если обнаружите синтаксические или другие ошибки, сверьте ваш исходный код с приведенным в книге. Убедитесь в том, что вы набрали все правильно. Внимательно следите за различными мелочами — круглыми скобками, двойными кавычками, наклонными чертами влево, знаками процентов, брызгами от чихов и другими необычными явлениями на вашем экране.

Если необходимо исправить ошибки, сделайте это прямо сейчас. Если нет, переходите к изучению следующего раздела.



- ✓ Об исправлении ошибок и повторной компиляции более подробно рассказано в главе 2 “Ошибки в программах на C”.
- ✓ Распространенная ошибка новичков: несоответствие двойных кавычек (компилятор сгенерирует сообщение вроде *Unmatched double quotes!*). Убедитесь в том, что вы используете двойные кавычки `"` парами. Если вы одну из кавычек пропустите, будет выведено сообщение об ошибке. Кроме того, убедитесь, что круглые и фигурные скобки также составляют пары, причем в каждой паре сначала идет левая скобка, а потом — соответствующая ей правая.

Награда

Хватит ждать! Запустите программу WHORU прямо сейчас. Наберите **whoru** или **/whoru** в командной строке и нажмите клавишу `<Enter>`. Вывод будет выглядеть следующим образом:

What is your name?

(Как Вас зовут?)

Программа теперь ждет, пока вы наберете свое имя. Вперед: наберите его! Нажмите `<Enter>`.

Если вы набрали **Buster**, следующая строка на экране будет такой:

Darn glad to meet you, Buster!

(Очень рад встрече с вами, Бастер¹!)

- ✓ Если вывод выглядит иначе, или программа работает неправильно, или выводится сообщение об ошибке, еще раз проверьте исходный код. Исправьте ошибки и снова откомпилируйте программу.

¹ Имена автор выбирает, конечно, те, которые были модными на диком Западе задолго до появления там арифмометров. Например, Бастер — Обездчик Лошадей. — Прим. ред.

- ✓ Ввод-вывод — это то, что компьютер делает лучше всего.
- ✓ Это пример программы, которая вводит информацию (имя) и генерирует вывод. Хотя данная программа лишь отображает введенную информацию (и никак иначе ее не обрабатывает), данная программа все равно относится к программам ввода-вывода.

Для ввода и вывода информации в исходной программе WHORU.C используются две мощные функции языка C: `printf()` и `scanf()`. В оставшейся части этой главы подробнее рассказывается об этих распространенных полезных функциях.

Подробнее о `printf()`

Функция `printf()` используется в языке программирования C для отображения информации на экране. Это команда вроде “Эй, я хочу кое-что сказать пользователю”, которая отображает текст на экране. Она годится на все случаи жизни, это своеобразный универсальный электрический карандаш языка C.

Формат использования базовой функции `printf` следующий:

```
printf("текст");
```

Слово `printf` всегда пишется в нижнем регистре. Обязательно. Далее следуют круглые скобки, в которых содержится строка текста в кавычках, в примере эта строка текста обозначена словом *текст*. Задача `printf()` состоит в том, чтобы отобразить *текст* на экране.

В языке C `printf()` — законченная инструкция. Точка с запятой всегда следует за последней круглой скобкой. (Конечно, есть исключения, но не стоит усложнять все на этом этапе.)

- ✓ Хотя *текст* и заключен в двойные кавычки, они не являются частью сообщения, которое `printf()` отображает на экране.
- ✓ Следует придерживаться определенных правил записи отображаемого текста. Все эти правила приведены в главе 24 “Глава о функции `printf()`”.
- ✓ Рассмотренный в предыдущем примере формат является упрощенным. Более полный формат `printf()` будет приведен в этой главе далее.

Печатаем прикольный текст

Дамы и господа, взгляните:

```
Ta da! I am a text string.
```

(Та там! Я — строка текста.)

Это просто набор текста, цифр, букв и других знаков, но это не строка текста. Нет. Чтобы эти символы рассматривались как целое, их нужно аккуратно заключить в двойные кавычки:

```
"Ta da! I am a text string."
```

("Та там! Я — строка текста.")

Теперь это строка текста. Но в программе она по-прежнему не является чем-либо осмысленным, ведь компьютер не может ее обработать. Чтобы компьютер мог обработать строку, ее нужно заключить в круглые скобки:

```
("Ta da! I am a text string.")
```

("Та там! Я — строка текста.")

Кроме того, для обработки строк нужен инструмент, или механизм — *функция*. Поставьте `printf` с одной стороны и точку с запятой с другой:

```
printf("Ta da! I am a text string.");
```

Получившийся бутерброд — команда языка C, которая просто отображает на экране текст, состоящий из цифр, букв и других символов. Просто и понятно.

Рассмотрим такую цепочку символов:

```
He said, "Ta da! I am a text string."
```

(Он сказал, "Та там! Я — строка текста.")

Это преступно или нет? Вроде бы это строка текста, но она содержит символы двойных кавычек. Можно ли сделать из этого текста строку, добавив дополнительные двойные кавычки?

```
"He said, "Ta da! I am a text string.""
```

("Он сказал, "Та там! Я — строка текста.""")

Итак, всего у нас четыре двойные кавычки, или восемь штрихов, парящих над строкой. Как C справляется с ними?

```
"Дамоклов меч", если я не ошибаюсь."
```

Компилятор языка C никогда не наказывает за испытание чего-либо. Может быть, вы думаете, что в Скалистых горах выдолблена большая пещера, в которой сидит маленький человечек и смотрит на миллионы экранов, один из которых отображает то же, что и монитор вашего ПК? И каждый раз, когда компилятор генерирует сообщение об ошибке, этот маленький человечек злобно хихикает? Ничего подобного нет и быть не может! И потому ошибки не опасны! Так почему бы не поэкспериментировать?

Пожалуйста, наберите следующий исходный код `DBLQUOTE.C`. Это еще один пример программы типа "`printf()` что-то отображает". Но теперь то, что отображается, содержит двойные кавычки. Получилось? Эта исходная программа — ваш эксперимент дня:

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

```
int main() // главная функция
{
    // ("Он сказал, "Ta da! Я текстовая строка.""")
    printf("He said, "Ta da! I am a text string.");
    return(0);
}
```

Наберите исходный код именно так, с двойными кавычками, которых всего четыре. (Вы сразу же заметите, что что-то не так, если ваш редактор кодирует текст разными цветами. Но чтобы разобраться с ошибками, сделайте так, как я показал в этом тексте.)

Сохраните файл с исходным кодом на диске под именем `DBLQUOTE.C`.

Откомпилируйте и запустите эту программу, если сможете. Скорее всего, вы столкнетесь со следующими ошибками:

```
dbquote.c: In function 'main':
```

```
dbquote.c:5: parse error before "Ta"
```

```
(dbquote.c: В функции 'главная':
```

```
dbquote.c:5: синтаксическая ошибка перед "Ta")
```

или

```
dbquote.c: In function 'main':
```

```
dbquote.c:6: syntax error before "Ta"
```


Строку текста, отображаемую функцией `printf()`, нужно заключить в двойные кавычки. Компилятор об этом знает. После второй двойной кавычки (в строке перед словом *Ta*) компилятор ожидал что-то другое, что-то отличное от "Ta". Поэтому было выдано сообщение об ошибке. В несколько карикатурной форме это проиллюстрировано на рис. 4.1.

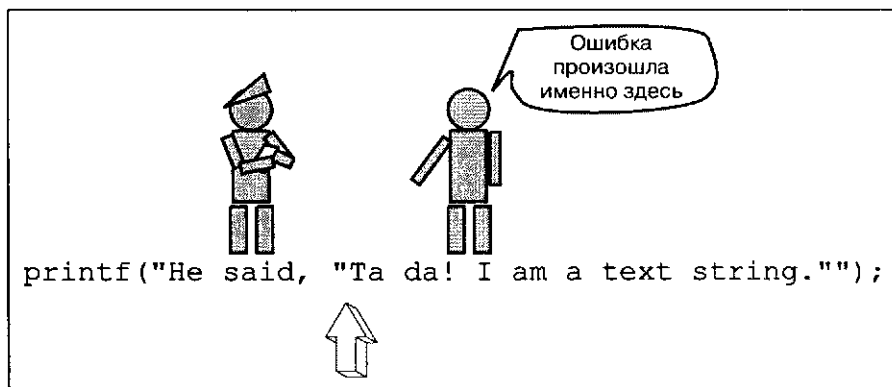


Рис. 4.1. Компилятор с языка C обнаружил что-то неладное в операторе `printf()`

Понятно, что иногда бывает нужно использовать в тексте символ двойных кавычек. Вопрос в том, как поместить его в `printf()`, не испортив себе остаток дня. Ответ: нужно использовать *escape-последовательность*.



Раньше программисты просто мирились с отсутствием некоторых символов. Чтобы не спотыкаться на двойных кавычках, они в строке везде вместо двойной кавычки ставили две одинарные. Некоторые древние программисты, которые не знают об *escape-последовательностях*, все еще используют подобные приемы.

Как обмануть `printf()`, или *escape-последовательности*

Escape-последовательности были придуманы, чтобы оживить описанную скучную картину несколькими яркими безрассудными штрихами. В языке программирования *escape-последовательности* позволяют незаметно протащить в текст запрещенные символы или символы, которые нельзя набрать непосредственно с клавиатуры.

В языке C *escape-последовательности* всегда начинаются с символа наклонной черты влево (`\`). Найдите этот символ на клавиатуре. Он должен быть над клавишей `<Enter>`, хотя его часто прячут в другие места.

Символ наклонной черты влево сигнализирует функции `printf()` о том, что вдали виднеется *escape-последовательность*. Когда `printf()` видит обратную косую, она думает: "О Боже, наверняка приближается *escape-последовательность*". И берет себя в руки, чтобы принять символ, который обычно она бы и на порог не пустила.

Чтобы незаметно протащить символ двойной кавычки, не взволновав `printf()`, нужно использовать *escape-последовательность* `\"` (наклонная черта влево, двойная кавычка). Вот новая, исправленная строка:

```
printf("He said, \"Ta da! I am a text string.\");
```

```
(printf("Он сказал, \"Та там! Я — строка текста.\");)
```

Обратите внимание на *escape-последовательности* `\` в строке текста. Они соответствуют двум двойным кавычкам, содержащимся внутри строки. Две внешние двойные кавычки, которые на самом деле являются границами строки, остаются нетронутыми. Странно, но ошибок нет.

(Если ваш текстовый редактор отображает текст программы различными цветами, будет видно, что `escape`-последовательности для двойных кавычек являются специальными символами, а не обычными кавычками, которые обозначают начало и конец строки.)

Отредактируйте файл вашей исходной программы `DBLQUOTE.C`. Внесите изменения в строку так, как показано выше. Нужно лишь вставить два символа наклонной черты влево перед этими шаловливыми двойными кавычками: `\`.

Сохраните измененную исходную программу на диске, заменив первоначальный файл `DBLQUOTE.C` новой версией.

Скомпилируйте и выполните. Теперь программа работает и выдает следующий текст:
`He said, "Ta da! I am a text string."`

(Он сказал, "Та там! Я строка текста.")



- ✓ `Escape`-последовательность `\` порождает символ двойной кавычки в середине строки.
- ✓ Еще одной полезной `escape`-последовательностью является `\n`, которая разбивает строку. Ввести символ перехода на новую строку простым нажатием клавиши `<Enter>` в строку текста нельзя, поэтому нужно использовать `escape`-последовательность `\n`.
- ✓ Все `escape`-последовательности начинаются с символа наклонной черты влево.
- ✓ Как вставить символ наклонной черты влево в строку? Используйте два таких символа. В строке `escape`-последовательность `\\` обозначает символ наклонной черты влево.
- ✓ В текстовой строке `escape`-последовательность может находиться где угодно: в начале, в середине или в конце, и столько раз, сколько вам нужно. По существу, знак `\` предназначен для вставки запрещенных символов в любую строку.
- ✓ Другие `escape`-последовательности приведены в главе 24 "Глава о функции `printf()`" (см. табл. 24.1).

`f` означает "formatted" ("форматированный")

Функция `printf()` названа так неспроста: *f* означает *formatted* (*форматированный*). Преимуществом функции `printf` над другими функциями C, отображающими тот или иной текст на экране, является возможность форматирования вывода.

Ранее в этой главе формат основной функции `printf` был представлен в виде:

```
printf("текст");
```

Но настоящий формат — это большой секрет! — вот такой:

```
printf("format_string" [, var[...]]);
```

То, что находится в двойных кавычках, является *строкой форматирования*. Это все еще текст, который появляется в выводе `printf()`, но в этот текст тайно вставлены различные *символы преобразования*, или специальные "заполнители". Они указывают функции `printf()` как форматировать выводимый текст и делать другие весьма впечатляющие вещи.

После строки форматирования идет запятая (все еще внутри скобок) и один или несколько элементов, называемых *параметрами*. Параметр, показанный в предыдущем примере, — это *var*, сокращение от *variable* (англ. "переменная"). Функцию `printf()` можно использовать для отображения содержимого (т.е. значения) одной или нескольких переменных. Это делается при помощи специальных символов преобразования в `format_string`. Как это происходит, показано на рис. 4.2.

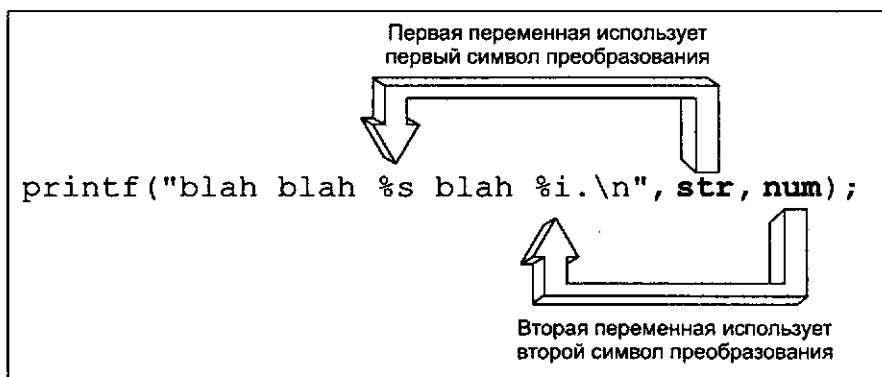


Рис. 4.2. Обработка параметров функцией `printf()`

Непонятная штука `[, ...]` означает, что в одной функции `printf` (перед закрывающей круглой скобкой) можно задать любое количество переменных, первая из которых обозначена *var*. Однако каждой переменной в *format_string* должен соответствовать заполнитель (или символ преобразования). Таким образом, количество переменных должно совпасть с количеством заполнителей (т.е. с количеством символов преобразования). В противном случае может произойти ошибка.

Немного выравнивания

Продemonстрируем, как с помощью `printf()` можно форматировать текст, и как использовать все те полезные *символы преобразования* и переменные *var*, которыми я вас пугал в предыдущем разделе. Как насчет коротенькой программы-примера?

Вам на рассмотрение предлагается следующая исходная программа, которую я назвал `JUSTIFY.C`. Полюбуйтесь:

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

```
int main() // главная функция
{
    printf("%15s", "right\n");
    printf("%-15s", "left\n");
    return(0);
}
```

Программа `JUSTIFY.C` делает следующее. Она отображает две строки: `right`, причем выравнивает строку с этим словом по правому краю (*right-justified*) и `left`, причем выравнивает строку с этим словом по левому краю (*left-justified*). Вы лучше поймете смысл этих терминов, когда увидите вывод программы, а не ее исходный код.

Наберите эту исходную программу в текстовом редакторе.

В первом операторе `printf` первой строкой является: `%15s` (знак процента, 15, строчная (малая) буква *s*). Затем идет запятая и `right`, а потом — *escape-последовательность*, обозначающая символ новой строки: `\n` (наклонная черта влево, строчная (малая) буква *n*).

Второй оператор `printf` почти такой же, только со знаком “минус” перед 15 и строкой `left` вместо `right`.

В этой программе больше всяких мелочей языка *C*, чем в любой другой из представленных в первых трех главах этой книги. Следите за тем, что набираете! Если вы уверены, что все набрано правильно, сохраните файл на диске под именем `JUSTIFY.C`.

Откомпилируйте JUSTIFY.C. При необходимости исправьте ошибки. Затем запустите программу. Вывод должен быть примерно следующим:

```
right
left
(        вправо
влево)
```

Слово `right` выровнено по правому краю в поле из 15 символов; `left` выровнено по левому краю. Такое расположение задается командой форматирования `%15s` в `printf()`. Часть строки форматирования `%15s` вовсе не печаталась. Вместо этого она управляла отображением на экране остального содержимого строки. Это реальный пример применения средств форматирования, предусмотренных в функции `printf()`.

В программе JUSTIFY.C есть лишь намек на то, что может делать функция `printf()`. В этой функции предусмотрено такое огромное количество форматов чисел, что вы были бы поражены наповал, если бы я рискнул представить их в этой главе прямо сейчас.

- ✓ В функции `printf()` первый элемент в кавычках — строка форматирования. На этом месте может также стоять текст, который будет отображаться на экране без изменения.
- ✓ Знак процента имеет особый смысл для `printf()`. Он определяет *символ преобразования* (называемый также *меткой-заполнителем*, или просто *заполнителем*). Символ преобразования в строке форматирования указывает, как функция `printf` должна форматировать вывод.
- ✓ Символ преобразования `s` означает “string” (стринг, строка): `%s`.
- ✓ Число между `%` и `s` указывает *ширину* отображаемой строки текста. Таким образом, `%15s` указывает, что строку текста нужно отобразить в поле из 15 символов. Знак “минус” перед 15 означает выравнивание выводимой строки по левому краю.
- ✓ “Выравнивание по левому краю” звучит как термин текстового процессора (редактора). Это и есть форматирование!
- ✓ Функция `printf()` не обрезает (не укорачивает) строки, более длинные, чем ширина, указанная в заполнителе `%s`.
- ✓ Все эти правила, связанные с символами преобразования, могут показаться сложными. Уверяю вас, что их редко кто помнит. Зачастую даже опытным программистам приходится обращаться к справочникам по C и проделывать несколько тестов, чтобы узнать, что делает та или иная команда форматирования. Впрочем, сталкиваться со сложным форматированием на практике приходится очень редко, так что не паникуйте.

Коротенькая справка: символы преобразования

Чтобы понять, как `printf()` использует строку форматирования и параметры, откройте в текстовом редакторе файл с исходным кодом программы GOODBYE.C. Замените строку

```
// ("Он сказал, "Ta da! Я текстовая строка.")
printf("He said, "Ta da! I am a text string.");)
```

следующей:

```
// До свидания, жестокий мир!
printf("%s", "Goodbye, cruel world!\n");
```

Вызов функции `printf()` изменился, теперь он содержит строку форматирования и параметр.

Строкой форматирования является `%s`, это заполнитель для строки.

Параметром является текст: `"Goodbye, cruel world!\n"`.

Сохраните исходную программу под новым именем `BYE.C`. Откомпилируйте и запустите ее. Вывод будет таким же, как и раньше. Просто в этот раз для форматирования вывода в функции `printf()` был использован заполнитель `%s`.

А теперь измените эту же строку следующим образом:

```
// ("%s, %s %s\n", "До свидания", "жестокая", "мир!")  
printf("%s, %s %s\n", "Goodbye", "cruel", "world!");
```

Внимательно отредактируйте указанную строку, после редактирования она должна в точности совпасть с приведенной выше строкой. Полученная строка содержит три заполнителя для строки `%s` и три строки в двойных кавычках (эти строки разделены запятыми). Сохраните программу. Откомпилируйте и выполните. Вывод должен быть тем же самым.

(Если при компиляции возникла ошибка, возможно, вы поместили запятую *внутри* двойных кавычек, а не между ними.)

scanf читается "скан-эф"

Вывод без ввода это как янь без инь — все равно, что салат "Цезарь" без чеснока. В таком случае песня семи гномов звучала бы занудно, вроде "О-О-О" вместо веселенького "Ай-Оу, Ай-Оу, Ай-Оу" ("I/O, I/O"). Кроме того — и это самое страшное — без ввода компьютер только ругал бы вас. Это просто ужасно.

К счастью, в языке C предусмотрены многочисленные инструментальные средства для того, чтобы заставить компьютер слушать вас. Огромное количество команд предназначено для считывания информации с клавиатуры. Среди них и команды считывания отдельных символов, и всеми любимая функция `scanf()`. Функция `scanf()` позволяет схватить строку текста с клавиатуры и сохранить ее в приятных теплых лапках строковой переменной.

- ✓ Функция `scanf()` похожа на `printf()`. Она считывает текст с клавиатуры.
- ✓ Буква *f* в `scanf()` так же, как и в `printf()`, означает *formatted* (форматированный). Функцию `scanf()` можно использовать для считывания с клавиатуры специальным образом отформатированного текста. Однако в этой главе мы обойдемся без усложнений и будем использовать `scanf()` только для считывания строки текста.

Компилируем `scanf`

Для того чтобы заставить `scanf()` работать, нужно две вещи. Во-первых, необходимо отвести место для хранения вводимого текста. Во-вторых, нужно вызвать функцию `scanf`.

Место в памяти для хранения строки называется *строковой переменной*. Слово *строка* означает строку символов, т.е. текст. *Переменная* означает, что строка не обязана быть постоянной — она может быть такой, какой ее введет пользователь. Строковая переменная — место в памяти для хранения текста, используемого в программах. (Переменные подробно обсуждаются в главе 8 "Переменные в языке C".)

Когда есть переменная для хранения текста, необходимо вызвать саму функцию `scanf()`. Формат вызова отчасти похож на таинственный расширенный формат `printf()`, так что нет смысла напирать на ваши извилины, объясняя его здесь.

В следующем примере `scanf()` считывает что-то имя. Сначала нужно выделить место в памяти (создать переменную) для хранения имени:

```
char firstname[20];
```

Эта инструкция языка C выделяет место в памяти для хранения строки текста (похоже на создание сейфа для огромной суммы денег, которую вы мечтаете однажды получить). Далее, как и сейф, переменная при создании “пуста”; в ней ничего не содержится до тех пор, пока вы не поместите туда что-нибудь.

Вот как можно разбить на части предыдущую инструкцию:

- `char` — это ключевое слово языка C, которое указывает, что компилятор должен создать символьную переменную, т.е. нечто, способное хранить текст (а не числа).
- `firstname` — это название места в памяти для хранения текста. В исходной программе для обращения к переменной достаточно указать ее название (имя) `firstname`.
- `[20]` определяет размер строки; в данном случае строка может содержать до 20-ти символов. Иначе говоря, мы выделили место для хранения 20-ти символов и назвали эту область (эту *переменную*) `firstname`.
- Точка с запятой завершает инструкцию языка C.

Следующий шаг состоит в том, чтобы использовать функцию `scanf()` для считывания текста с клавиатуры и сохранения его в созданной переменной. Запишем нужный нам вызов: `scanf("%s",&firstname);`

Вот как работает эта инструкция:

- `scanf()` — функция для считывания информации с клавиатуры.
- `%s` — заполнитель для строки; `scanf()` ожидает ввода обычного текста с клавиатуры. Нажатие клавиши <Enter> заканчивает ввод.
- Введенный текст сохраняется в строковой переменной, которой мы дали название (имя) `firstname`. Амперсанд & необходим для того, чтобы функция `scanf()` могла найти строковую переменную в памяти.
- Точка с запятой завершает инструкцию языка C.

Прежде чем записать текст в переменную, функция `scanf()` считывает его с клавиатуры и сохраняет в памяти компьютера для дальнейшего использования. Пример будет приведен в следующем разделе.

- ✓ Если программа на языке C вводит информацию, необходимо сначала создать место для хранения этой информации. В случае ввода текста таким местом является строковая переменная. Она создается с помощью зарезервированного слова `char` (символ).
- ✓ Переменные будут официально представлены в главе 8 “Переменные в языке C” этой книги. Пока же используемую функцией `scanf()` строковую переменную можно считать просто местом для хранения вводимого текста.
- ✓ Коды форматирования, используемые в `scanf()`, совпадают с кодами, используемыми в `printf()`. В действительности их чаще используют в функции `printf()`, так как для считывания с клавиатуры существуют способы и полнее. В табл. 24.4 главы 24 “Глава о функции `printf()`” приводится список кодов форматирования (это те самые заполнители, которые начинаются со знака процента %).

- ✓ Часто встречающаяся ошибка — отсутствие символа & перед именем переменной в функции `scanf()`. Пропуск этого знака приводит к удивительным ругательствам вроде *null pointer assignment* (присваивание пустому указателю), которыми, возможно, вас наградит компилятор в дальнейшем. Однако сверхъестественная причуда компилятора состоит в том, что амперсанд & не является обязательным перед строковыми переменными. Кто бы мог подумать?!

Чудо `scanf()`

Рассмотрим следующую бессмысленную программу `COLOR.C`, в которой используются две строковые переменные, `name` (имя) и `color` (цвет). Программа просит ввести ваше имя, а затем — ваш любимый цвет. После этого заключительная инструкция `printf()` отображает на экране то, что вы ввели.

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

```
int main() // главная функция
{
    char name[20];
    char color[20];

    printf("What is your name?"); // Как вас зовут?
    scanf("%s", name); // имя
    // Какой у вас любимый цвет?
    printf("What is your favorite color?");
    scanf("%s", color); // цвет
    // (" %s's любимый цвет - %s\n ", имя, цвет)
    printf("%s's favorite color is %s\n", name, color);
    return(0);
}
```

Введите этот исходный текст с помощью вашего редактора. Сохраните этот файл на диске под именем `COLOR.C`. Скомпилируйте.

Если появятся сообщения об ошибках, перепроверьте ваш исходный текст и снова отредактируйте файл. При вводе этой программы часто встречается следующая ошибка: программист в заключительном операторе `printf()` пропускает одну (или даже обе!) из двух запятых.

Выполните программу! Вывод будет выглядеть примерно так:

```
What is your name? dan
What is your favorite color? brown
dan's favorite color is brown
```

(Как Вас зовут? Дэн

Каков ваш любимый цвет? коричневый цвет
любимый цвет Дэна коричневый)

В Windows XP эту программу можно запускать с помощью следующей командной строки:

```
.\color
```

Дело в том, что в Windows XP имя `COLOR` могут иметь и другие консольные команды, используемые, например, для изменения цвета переднего плана и фона консольного окна.

Время поэкспериментировать!

Что важнее: порядок таких мелочей, как `%s`, или порядок переменных (параметров) в функции `printf`? Сдаётся? Я не скажу ответ. Выяснить это придется вам самим.

Измените последнюю инструкцию печати в программе COLOR.C следующим образом:

```
// ("%s's любимый цвет - %s\n ", цвет, имя)
printf("%s's favorite color is %s\n", color, name);
```

Здесь изменен порядок переменных: сначала идет color (цвет), а потом — name (имя). Сохраните изменения на диске и откомпилируйте программу заново. Программа все еще работает, но вывод стал другим. Это потому, что изменился порядок переменных. На экране вы увидите следующее:

```
brown's favorite color is Dan.
```

(любимый цвет коричневого цвета — Дэн)

Видите? Компьютеры *на самом деле* глупы! Если в функции printf() используется несколько переменных, то важен их порядок. Знаки %s? Они лишь играют роль меток-заполнителей пустых мест, которые будут заполнены информацией, хранящейся в переменных. Иными словами, это нечто вроде пустых мест на чистом бланке.

А что будет после такой замены:

```
// ("%s's любимый цвет - %s\n ", имя, имя)
printf("%s's favorite color is %s\n", name, name);
```

В этой модификации инструкции печати переменная name используется дважды — это вполне допустимо. Ведь при вызове функции printf() нужно следить только за тем, чтобы в формирующей строке меткам-заполнителям %s соответствовали обращения к строковым переменным. В данном вызове обоим меткам-заполнителям %s соответствуют обращения к строковой переменной name. Все это законно. Сохраните эти изменения и перекомпилируйте программу. Выполните ее и изучите вывод:

```
Dan's favorite color is Dan
```

(Любимый цвет Дэна — Дэн)

Такую ошибку мог допустить только горький алкаш. Сделав такую ошибку в налоговом отчете, вам придется годами играть в гольф с бывшими биржевыми брокерами и конгрессменами. (Поэтому уже сейчас возьмите себе в привычку следить за порядком переменных в вызовах функций.)

Наконец, измените последнюю инструкцию печати в программе COLOR.C следующим образом:

```
// ("%s's любимый цвет - %s\n ", имя, "синий")
printf("%s's favorite color is %s\n", name, "blue");
```

Вместо переменной color использована *строковая константа*. *Строковая константа* — это просто строка, взятая в кавычки. Она не изменяется в отличие от переменной, которая может содержать что угодно. (Это ведь не переменная!)

Сохраните изменения на диске и заново скомпилируйте полученный исходный текст. Программа работает, но какой бы цвет вы не задавали, компьютер все время настаивает на том, что любимый цвет blue (синий).

- ✓ Строковая постоянная "blue" (синий) допустима потому, что заполнитель %s в строке форматирования функции printf() соответствует строке текста. (Он просто ищет строку текста.) Не имеет значения, как будет задана эта строка — в виде строковой переменной или в виде "настоящего" текста, заключенного в двойные кавычки. (Конечно, преимущество программ с вводом-выводом как раз и состоит в том, что для хранения вводимой информации можно использовать переменные. Использовать константы несколько неразумно, поскольку в этом случае компьютер уже и так "знает", что он будет выводить. Дамы и господа, позвольте задать вопрос: а зачем же тогда ввод-вывод?)



- ✓ Заполнитель `%s` в функции `printf()` ищет соответствующую строковую переменную и вставляет ее содержимое в текст, отображаемый на экране.
- ✓ Каждой метке-заполнителю `%s`, которая появляется в строке форматирования `printf()`, должна соответствовать строковая переменная в функции `printf()`. Если переменная отсутствует, компилятор может выдать сообщение о синтаксической ошибке.
- ✓ Вместо строковых переменных можно использовать строковые константы, часто называемые *строками-литералами*, или *литеральными* (буквальными) строками. Однако это нерационально, поскольку в этом случае впустую тратится время на операции с `%s`, несмотря на то, что вы заранее знаете, что будете отображать на экране. (Я вынужден был продемонстрировать здесь эту возможность — в противном случае меня могли бы выслать в штат Коннектикут, чтобы засадить там в специальную тюрьму для нерадивых преподавателей программирования на языке C.)
- ✓ Убедитесь в том, что переменные указаны в нужном порядке. Соблюдать порядок особенно важно в том случае, если вы используете в `printf` как числовые, так и строковые переменные.
- ✓ Знак процента (`%`) является священным. *О!* Если с помощью `printf` нужно вывести (отобразить) знак процента (`%`), укажите в строке форматирования два знака процента: `%%`.

Комментарии: С или не С

В этой главе...

- Вставка примечаний для себя
- Использование причудливых методов комментирования
- Заимствованные комментарии в стиле C++
- Отключение кода с помощью комментариев
- Избегайте вложенных комментариев

В программировании очень важно не забыть, что вы делаете. Я не говорю о программировании непосредственно — об этом-то помнить просто, и, кроме того, вы можете покупать книги и справочники в изобилии в случае, если вы не хотите помнить синтаксические правила. Вместо этого вы должны помнить, *что* вы пытаетесь программу заставить сделать в определенной команде. Чтобы не забыть это, вставляйте комментарии в ваш исходный текст.

Комментарии на самом деле не так уж необходимы в маленьких программах, которые встречаются в этой книге. Комментарии не столь уж необходимы до тех пор, пока вы не начнете писать большие программы, например, такие как Excel или Photoshop, в которых легко потерять ход мысли. Чтобы напомнить себе о том, что вы делаете, необходимо вставить комментарий в исходный текст, чтобы объяснить ваш подход к решению задачи. С помощью комментариев исходный текст можно бегло просмотреть снова, при этом не нужно будет пристально вглядываться в каждый символ кода программы, потому что комментарии напомнят вам о том, что именно закодировано в программе.

Добавление комментариев

Комментарий в программе на языке С имеет начало и конец. Все между началом и концом комментария игнорируется компилятором, а это означает, что туда можно вставить любой текст и он не повлияет на выполнение программы.

/ Вот как выглядит комментарий на языке С */*

Приведенная выше строка — прекрасный пример комментария. Ниже приведен еще один пример комментария, причем комментарий этого типа обычно используются для повышения репутации:

```
/*
Привет, компилятор! Эй, тут ошибка: printf!
Ха! Ха! Все равно ты не видишь меня! Ух!
Вот тебе за твои придирки! Вот тебе за твои придирки!
Вот тебе за твои придирки!
*/
```

- ✓ В начале комментария ставится наклонная черта вправо и звездочка: /*.
- ✓ В конце комментария ставится звездочка и наклонная черта вправо: */.
- ✓ Начало и конец комментария отличаются.
- ✓ Комментарий — это не инструкция языка C. После /* точка с запятой не нужна.

Большая, запутанная программа с комментариями

Следующий исходный текст — MADLIB1.C. Это большая, запутанная программа с комментариями. В ней используются функции printf() и scanf(), описанные в главе 4 “Что такое ввод-вывод?”, чтобы создать короткую и все же интересную историю.

```
/*
MADLIB1.C Исходный текст
Написана (ваше имя здесь)
*/

#include <stdio.h>

int main()                                // главная функция
{
    char adjective[20];                    // прилагательное (20 символов)
    char food[20];                         // продовольствие (20 символов)
    char chore[20];                        // хозяйственная работа (20 символов)
    char furniture[20];                    // мебель (20 символов)

    /* Ввести слова в madlib */

    printf("Enter an adjective:");         /* запрос */
    scanf("%s",&adjective);                /* ввод */
    printf("Enter a food:");
    scanf("%s",&food);                     // Введите продовольствие
    printf("Enter a household chore (past tense):");
    // Введите домашнюю хозяйственную работу (прошедшее время)
    scanf("%s",&chore);
    printf("Enter an item of furniture:");
    scanf("%s",&furniture);                // Введите элемент мебели

    /* Отобразить вывод */
    printf("\n\nDon't touch that %s %s!\n",adjective,food);
    printf("I just %s the %s!\n",chore,furniture);
    // Не прикасайся к этому %s %s!\n, прилагательное, продовольствие
    // я только %s %s!\n, хозяйственная работа, мебель

    return(0);
}
```

Напечатайте исходный текст точно так, как написано¹. Единственная новая особенность этой программы — комментарии. Каждый комментарий начинается с /* и заканчивается */. Удостоверьтесь, что они набраны правильно: наклонная черта вправо со звездочкой начинает комментарий, а звездочка с наклонной чертой (тоже вправо) заканчивает его. (Если редактор кода использует выделение цветом, все комментарии будут выделены одним цветом.) Сохраните файл на диске и назовите его MADLIB1.C.

Скомпилируйте и выполните.

Ниже приведен пример вывода программы:

¹ Комментарии, начинающиеся с двух косых, добавлены редактором русского перевода. Их можете не набирать. — Прим. ред.

Enter an adjective: hairy
Enter a food: waffle
Enter a household chore (past tense): vacuumed
Enter an item of furniture: couch

Don't touch that hairy waffle!
I just vacuumed the couch!

О, ха-ха! Ну и челуха! Ну и история!

- ✓ Эта программа длинная и выглядит сложной, но в ней нет никаких новых уловок. В ней все уже вам знакомо: `char` создает строковые переменные, `printf()` отображает текстовые и строковые переменные, а `scanf()` читает то, что вводится с клавиатуры. Можете зевнуть.
- ✓ В `MADLIB1.C` используются следующие четыре строковые переменные: `adjective` (прилагательное), `food` (продовольствие), `chore` (хозяйственная работа) и `furniture` (мебель). Все четыре переменные создаются с помощью ключевого слова `char` и для каждой из них в памяти отводится 20 символов. Каждая строковая переменная заполняется функцией `scanf()`, когда вы выполняете ввод с клавиатуры.
- ✓ Каждый из расположенных в конце вызовов функции `printf()` содержит два символа форматирования `%s`. Две строковые переменные в каждом вызове функции хранят текст для символов форматирования `%s`.
- ✓ Предпоследняя функция `printf()` начинается с двух символов новой строки `\n \n`. Эти символы отделяют раздел ввода данных, в котором вы вводите текст, от раздела вывода программы. Да, новые строки могут появиться в любом месте строки, а не только в ее конце.
- ✓ В программе `MADLIB1.C` пять комментариев². Удостоверьтесь, что вы можете найти каждый. Обратите внимание, что они все разные, хотя каждый начинается с `/*` и заканчивается `*/`.

Зачем нужны комментарии?

Комментарии для компилятора с C не являются необходимыми. Он игнорирует их. Комментарии предназначены для программиста. Они содержат советы, предположения о том, что вы пробуете сделать, или описывают работу программы. В комментариях можно написать все, что угодно, хотя чем более он информативный, тем полезнее он будет вам впоследствии.



Большинство программ на C начинается с нескольких строк комментариев. Все мои программы на C начинаются с информации следующего типа:

```
/* COOKIES.C  
Дэн Гукин, 1/20/05 2:45 утра.  
Scan Internet files for expired dates and delete.  
Сканирует cookie-файлы Internet с истекшими датами и удаляет их.  
*/
```

В этих нескольких строках указано, зачем написана программа и когда я начал работать над ней.

В самом исходном тексте можно писать комментарии в виде примечаний, заметок для себя и т.д., например, можно написать такой комментарий:

```
/* Узнать, почему это не работает */
```

² Не считая, конечно, вставленных редактором русского перевода. — Прим. ред.

или такой:

```
save=itemv; /* Сохранить старое значение здесь */
```

или даже напоминания себе на будущее:

```
/*  
Когда-нибудь сюда надо будет вставить код, который заставляет  
компьютер запомнить момент последнего запуска этой программы.  
*/
```

Дело в том, что комментарии — примечания для программиста. Если бы вы изучали программирование на С в институте, вы научились бы писать комментарии так, чтобы удовлетворить ваших профессоров. Если вы участвуете в разработке большого программного продукта, комментарии умиротворяют вашего руководителя группы. Комментарии в программах предназначены для вас.

Стили комментариев: профессиональный и будущих профессионалов (пока еще любителей)

Программа MADLIB1.C содержит пять комментариев трех различных стилей комментирования. Вы можете изобрести еще много способов комментирования ваших программ, однако имеющиеся в MADLIB1.C встречаются чаще всего.

```
/*  
MADLIB1.C Исходный текст  
Написан Майком Роусофтом  
*/
```

Здесь показан самый популярный многострочный комментарий. Первая строка, в которой начинается комментарий, содержит только /*. После нее все строки комментария, что бы они не содержали — замечания, напоминания или объяснения, — игнорируются компилятором. Последняя строка, в которой заканчивается комментарий, содержит только */. Не забудьте, что в конце комментария нужно поставить */; иначе компилятор с языка С подумает, что вся ваша программа состоит только из одного большого, длинного комментария (сделать такое возможно, но не рекомендуется).

```
/* Ввести слова в madlib */
```

Эта строка — однострочный комментарий, не перепутайте его с инструкцией языка С. Комментарий начинается с /* и заканчивается */, причем весь он умещается на той же самой строке. Поскольку это комментарий, а не инструкция, точка с запятой не нужна.

Наконец, вы можете добавить комментарий в конце строки:

```
printf ("Enter an adjective:"); /* запрос */
```

После инструкции printf и нескольких символов табуляций начинается комментарий с /*, причем заканчивается он */ на той же самой строке.

Суперкомментарии

Путешествуя по программам, я видел много попыток придать с помощью комментариев привлекательный вид программам на С. Вот пример такого комментария:

```

/*****
** Превосходная программа командующего Зеро **
*****/

```

Это действительно комментарий. Он содержит много звездочек, но все они находятся между /* и */, а потому это вполне жизнеспособный комментарий.

Ранес я использовал в моих программах вот что:

```

/*
* Это — многоречивое введение в
* непонятную программу, написанную кем-то в
* университете, кто слишком много думает о себе и
* к тому же уверен, что ни один простой смертный
* не может изучить С -- и кто
* написал три книги по С, чтобы доказать это.
*/

```

Идея в этом примере состоит в том, чтобы создать “стену звездочек” между /* и */, и тогда комментарий кажется приклеенным к странице.

Вот еще один пример часто используемого мною стиля:

```

/*****

```

Эта строка звездочек не содержит какого-либо осмысленного текста, но все же она помогает разделять различные разделы программы. Например, я могу поместить строку звездочек между различными функциями, чтобы их можно было легко найти.



Резюме: независимо от того, что находится между строками, комментарий должен начинаться с /* и в конце его должны быть символы */.

Комментарии в C++

Поскольку сегодняшние компиляторы с языка С также создают код с C++, в ваших простых программах на старом С можно использовать комментарии в стиле C++³. Я упоминаю это просто потому, что комментарии в стиле C++ могут быть тоже полезны, и потому обычно они допускаются и в программах на С⁴.

В C++ комментарии могут начинаться с двух наклонных черт вправо, т.е. с //. Они указывают, что остальная часть текста в строке — комментарий. Конец строки отмечает конец комментария:

```

// Это --- другой стиль комментария,
// стиль, используемый в C++

```

Этот стиль комментариев имеет то преимущество, что вы не забудете закончить комментарий, поэтому он идеально подходит для того, чтобы помещать такие комментарии в конце инструкции языка С, как показано в следующем примере:

```

                                // Введите прилагательное
printf("Enter an adjective:");    // запрос
scanf("%s",&adjective);          // ввод

```

Такие модификации программы MADLIB1.C не затрагивают смысл комментария. Этот метод предпочтителен, потому что более легкий для ввода; однако комментарий с /* и */ имеет преимущество, если нужно вставить длинный текст, поскольку в этом случае не придется повсюду печатать //.

³ Именно этот стиль чаще всего используется в комментариях, добавленных редактором русского перевода. — Прим. ред.

⁴ Если, конечно, вы пользуетесь компилятором C++. — Прим. ред.

Комментарии как средство отключения кода

Комментарии игнорируются компилятором. Независимо от того, какая чепуха (или даже наглая ложь) помещается между /* и */, компилятор через это перескакивает. Любая информация (включая суммы наличных денег, рецепты вечной молодости) игнорируется, если она помещена внутри комментария.

Измените исходный текст MADLIB1.C, точнее измените только последнюю часть программы, чтобы она выглядела так⁵:

```
/* Отобразить вывод */
/*
printf("\n\nDon't touch that %s %s!\n", adjective, food);
printf("I just %s the %s!\n", chore, furniture);
// Не прикасайся к этому %s %s!\n, прилагательное, продовольствие
// я только %s %s!\n, хозяйственная работа, мебель
*/
```

Чтобы модифицировать программу, выполните следующие шаги:

1. Вставьте строку с /* перед первым вызовом функции printf() в этом примере.
2. Вставьте строку с */ после второго вызова функции printf().

Заблокировав таким образом последние две инструкции printf() языка C, сохраните файл на диске, и перетранспируйте его. Он выполняется, как и прежде, но в конце “безумные сообщения” не отображаются. Причина состоит в том, что заключительные два вызова функции printf() в конце программы “закомментированы”.



- ✓ С помощью комментариев можно отключить некоторые части программы. Если что-то работает неправильно, его можно “закомментировать”. Кроме того, можно также вставить примечание, чтобы объяснить, почему данный раздел закомментирован.
- ✓ Иногда случается так, что кое-что, что должно работать, не работает. Причина, возможно, заключается в том, что вы случайно закомментировали эту часть программы. Всегда проверяйте метки комментариев /* и */, чтобы удостовериться, что они стоят именно там, где нужно.
- ✓ Если редактор кода размечает текст цветом, можно легко определить отсутствие символов */, которые заканчивают комментарий. Если слишком большая часть исходного текста окрашена в цвет комментария, всему виной, вероятно, отсутствие */.

Опасности вложения комментариев

Наибольший грех, в который вас могут ввести комментарии, — это так называемые “вложенные” комментарии. Вложенным называется комментарий, находящийся внутри другого комментария. Рассмотрим следующий фрагмент программы на C:

⁵ Как всегда, комментарии, начинающиеся с двух косых, добавлены редактором русского перевода. Их можете не набирать. Впрочем, теперь вы суперпрофессионал по части комментариев и поэтому в дальнейшем я подобные примечания опускаю — ведь вы во всем этом уже разбираетесь сами. — Прим. ред.


```

if(all_else_fails)
{
    display_error(errno);          /* errno уже установлен */
    walk_away();
}
else
    get_mad();

```

Не волнуйтесь, если сейчас не понимаете суть этого примера; она станет понятной позже. Однако обратите внимание, что после вызова функции `display_error` следует комментарий: `errno` уже установлен. Но предположим, что вы хотите изменить эту часть программы так, чтобы выполнялась только функция `get_mad()`. Нужно закомментировать все, за исключением той строки, которая должна выполняться:

```

/*
if(all_else_fails)
{
    display_error(errno);          /* errno уже установлен */
    walk_away();
}
else
*/
get_mad();

```

Здесь компилятор с языка С видит только функцию `get_mad`, правильно?

Неправильно! Комментарий начинается на первой строке символами `/*`. Но он заканчивается в строке с функцией `display_error()`. Поскольку эта строка заканчивается символами `*/` — признаком конца комментария, это и есть конец комментария. Поэтому компилятор с языка С снова начинает с функции `walk_away` и генерирует синтаксическую ошибку, когда видит фигурную скобку, свободно плавающую в море пробелов. Второй признак конца комментария (чуть выше функции `get_mad()`) также приводит к обнаружению ошибки. Две ошибки! Как отвратительно!

В этом примере показан вложенный комментарий, или комментарий внутри другого комментария. Так использовать комментарии нельзя⁶. На рис. 5.1 показано, как компилятор с языка С интерпретирует вложенный комментарий.

Чтобы избежать западни вложенных комментариев, нужно быть внимательным при отключении частей программы на С. Решение в данном случае состоит в том, чтобы раскомментировать комментарий `errno` уже установлен. Или же можно закомментировать каждую отдельную строку, тогда рассматриваемая строка выглядела бы следующим образом:

```

/*    display_error(errno);    /* errno уже установлен */

```

Этот метод работает потому, что комментарий все равно заканчивается символами `*/`. Дополнительные символы `/*` в комментарии благополучно игнорируются.

- ✓ Да, вложенных комментариев компилятор не любит, но пока нечего волноваться о них.
- ✓ Обратите внимание, что для комментариев в стиле С++, начинающихся с `//`, нет проблемы вложения.

⁶ Некоторые компиляторы допускают вложенные комментарии. — Прим. ред.

Вот что написано:

```
/*
if(all_else_fails)
{
    display_error(errno); /* errno уже установлен */
    walk_away();
}
else
*/
    get_mad();
```

А вот что видит компилятор:

```
/*
if(all_else_fails)
{
    display_error(errno); /* errno уже установлен */
    walk_away();
}
else
*/
    get_mad();
```

Ошибка!

Еще одна ошибка!

И еще много других ошибок!

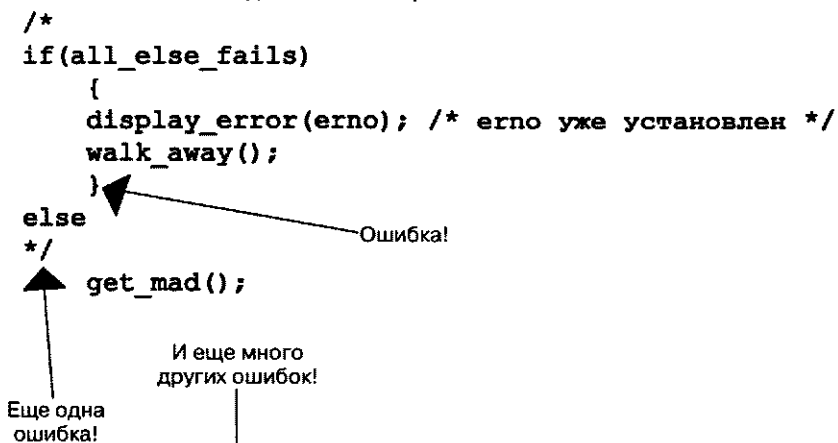


Рис. 5.1. Опасности, связанные с вложенным комментарием

Ввод-вывод с помощью функций `gets()` и `puts()`

В этой главе...

- Чтение строк текста с помощью `gets()`
- Как избежать некоторых проблем, связанных с `gets()`
- Использование `puts()` для отображения текста
- Отображение переменных с помощью `puts()`
- Что использовать: `puts()` или `printf()`

К функциям ввода-вывода, т.е. к функциям, отображающим информацию на экране или считывающим текст с клавиатуры, относятся не только `printf()` и `scanf()`. Нет, язык C полон уловок ввода-вывода, и когда вы узнаете обо всех ограничениях и недостатках `printf()` и `scanf()`, вы, вероятно, создадите ваши собственные функции, которые будут считывать символы с клавиатуры и отображать информацию так, как вы того захотите. Но пока вы должны пользоваться тем, что предлагает C.

В этой главе вводятся простые функции `gets()` и `puts()`. Функция `gets()` читает строку текста с клавиатуры, а `puts()` отображает строку текста на экране.

Функция `gets()`: чем больше я хочу, тем больше я получу

По сравнению со `scanf()` функция `gets()` намного лучше и проще. Обе функции делают то же самое. Они читают символы с клавиатуры и сохраняют их в переменной. Но `gets()` читает только текст. Функция `scanf()` может читать числовые значения и строки, притом в самых различных комбинациях. Иногда это действительно нужно, но если требуется считать только текст, применение этой функции выглядит неуклюже.

Подобно `scanf()`, функция `gets()` сохраняет считанный текст в строковой переменной. Эта функция читает все, что печатается на клавиатуре до нажатия клавиши <Enter>. Вот формат вызова:

```
gets(var);
```

За `gets()`, как и за всеми остальными функциями, следует пара круглых скобок. Поскольку `gets()` — законченная инструкция, она всегда заканчивается точкой с запятой. В круглые скобки заключается переменная, т.е. название (имя) строковой переменной, в которой функция запоминает текст.

Пример программы

Вот пример программы, в которой используется `gets()`, — программа `INSULT1.C`. Эта программа почти идентична программе `WHORU.C`, рассмотренной в главе 4 “Что такое ввод-вывод?”, за исключением того, что используется `gets()`, а не `scanf()`.

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

```
int main() // главная функция
{
    char jerk[20];

    printf("Name some jerk you know:");
    gets(jerk);
    printf("Yeah, I think %s is a jerk, too.\n",jerk);
    return(0);
}
```

Введите этот исходный текст с помощью вашего редактора. Сохраните файл на диске и назовите его `INSULT1.C`.

Скомпилируйте программу. Если будут обнаружены ошибки, заново отредактируйте текст. Не забудьте поставить точки с запятой и обратите внимание на использование двойных кавычек в функции `printf()`.

Выполните программу. Вывод будет выглядеть примерно так:

```
Name some jerk you know:Bill
Yeah, I think Bill is a jerk, too.
```

- ✓ Функция `gets()` читает текст в переменную точно так же, как это делает `scanf()`. Но независимо от того, чем прочитан текст, его можно отобразить с помощью инструкции `printf()`.
- ✓ Выражение `gets(var)` эквивалентно выражению `scanf("%s",var)`.
- ✓ Если при компиляции появились ошибки-предупреждения, прочтите следующий раздел.
- ✓ Имя `gets()` можно читать как “get-string” (“ввести строку текста с клавиатуры”). Однако это, вероятно, подразумевает “get stdin”, что означает “ввести со стандартного ввода”. Как бы то ни было, а программа получает строку.



Недостатки функции gets()



В последнее время в серьезных и безопасных программах на языке C не рекомендуется использовать функцию `gets()`. Дело в том, что `gets()` не считается безопасной функцией.

Предупреждение об этом может появиться даже при компиляции программы. Причина состоит в том, что пользователь может напечатать на клавиатуре больше символов, чем вмещается в строковой переменной, указанной в `gets()`. Этот недостаток, иногда называемый переполнением буфера, используется многими жуликами для написания “червей” и вирусов и изменения режима работы хорошо знакомых программ.

Читая эту книгу, не волнуйтесь о недостатках `gets()`. Она вполне подходит для целей этой книги, поскольку позволяет быстро ввести текст. Но для “реальных” программ я рекомендую придумать ваши собственные функции для чтения клавиатуры.

Достоинства функции puts()

В некотором смысле, функция puts() — упрощенная версия функции printf(). Функция puts() отображает строку текста, но безо всякого волшебства форматирования, присущего printf(). Функция puts() выполняет только тупую команду: отобразить текст на экране. Вот ее формат:

```
puts(текст);
```

После имени puts() ставится левая скобка, а затем — текст, который нужно отобразить (это может быть имя строковой переменной или строка текста в двойных кавычках). За текстом ставится правая скобка. Вызов функции puts() — законченная инструкция языка C, так что она всегда заканчивается точкой с запятой.

Вывод функции puts() всегда заканчивается символом \n (новая строка). Функция puts() как бы нажимает <Enter> после отображения текста. Избежать этого побочного эффекта невозможно, правда иногда именно это и требуется.

Приглашение к вводу команды: запускаем еще одну глупую программу

Чтобы разобраться, как работает функция puts(), создайте следующую программу, назовем ее STOP.C. Да, эта программа действительно глупа, но вы еще только начинаете изучение C, так что потерпите — осмысленные программы будут позже:

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

```
int main()                                // главная функция
{
    // Не могу остановиться: ошибка плохого настроения.
    puts("Unable to stop: Bad mood error.");
    return(0);
}
```

Сохраните этот исходный текст на диске под именем STOP.C. Скомпилируйте его, скомпонуйте и выполните.

Если после приглашения к вводу команды напечатать stop или ./stop (останов), эта программа генерирует следующий вывод:

```
Unable to stop: Bad mood error.
```

(Не могу остановиться: ошибка плохого настроения.)

Ха-ха!



- ✓ Как и printf(), puts() отображает строку текста на экране. Текст нужно заключить в двойные кавычки и все это поместить между двумя круглыми скобками.
- ✓ Как и printf(), puts() понимает управляющие последовательности. Например, чтобы отобразить строку с двойной кавычкой, можно использовать \".
- ✓ В конце строки текста, печатаемой с помощью функции puts(), помещать \n не нужно, поскольку puts() всегда отображает символ новой строки в конце своего вывода.
- ✓ Если символ новой строки в конце вывода не нужен, следует использовать не puts(), а printf().

Функции puts() и gets() в действии

Следующая программа — тонкая модификация INSULT1.C. На сей раз первый вызов printf() заменен инструкцией puts():

```
#include <stdio.h>

int main()                                // главная функция
{
    char jerk[20];
    puts("Name some jerk you know:");
    gets(jerk);
    printf("Yeah, I think %s is a jerk, too.",jerk);
    return(0);
}
```

Загрузите исходный текст INSULT1.C в редактор и замените printf на puts.

Используйте команду редактора Save As (Сохранить как), чтобы измененный исходный текст сохранить на диске под новым именем INSULT2.C. Сохраните. Скомпилируйте. Выполните.

```
Name some jerk you know:
Rebecca
Yeah, I think Rebecca is a jerk, too.
```

Обратите внимание, что после первой отображаемой функцией puts() строки выводится символ перехода на новую строку. Именно поэтому ввод происходит со следующей строки. Как видите, программа выполняет то же самое, что и INSULT1. Но не удивляйтесь и продолжайте читать дальше.

Еще одна модификация программы INSULT

Следующий исходный текст — еще одна модификация программы INSULT. На сей раз заключительную инструкцию printf() нужно заменить инструкцией puts(). Вот как выглядит новая модификация программы:

```
#include <stdio.h>

int main()                                // главная функция
{
    char jerk[20];

    puts("Name some jerk you know:");
    gets(jerk);
    puts("Yeah, I think %s is a jerk, too.",jerk);
    return(0);
}
```

Загрузите исходный текст INSULT2.C в редактор. Сделайте только что указанные изменения, главное из них — замена printf на puts. Остальная часть кода не изменяется.

Сохраните новый исходный текст на диске под именем INSULT3.C. Скомпилируйте и выполните.

Ужас! Ошибка! Ошибка!

```
Insult3.c:9: too many arguments to function 'puts'
(Insult3.c:9: слишком много параметров у функции 'puts')
```

Компилятор обратил внимание, что для функции puts() указано более одного параметра: сначала указана строка, а затем переменная. Но компилятор знает, что нужен только один параметр, а не два. Ой!

- ✓ Функция `puts()` только чуть более проста, чем `printf()`.
- ✓ Если все же удастся запустить программу (некоторые компиляторы могут создать исполнимый файл, несмотря на ошибку), вывод будет примерно таким:
 Name some jerk you know:
 Bruce
 Yeah, I think that %s is a jerk, too.

Кто же этот %s? Что это за человек, который является таким безобразником? Никто не знает! Помните, что `puts()` — не `printf()`, и эта функция не обрабатывает переменные так, как `printf()`. Для `puts()` %s в ее строке — только символы %s, а не что-то особенное.

Функция `puts()` может печатать переменные

Функция `puts()` может отобразить строковую переменную, но только в отдельной строке. Почему в отдельной строке? Потому что `puts()` всегда выводит тот противный символ новой строки. С помощью функции `puts()` нельзя поместить переменную в другую строку текста.

Рассмотрим следующий исходный текст, последний в серии программ INSULT:

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

```
int main()
{
    char jerk[20];

    puts("Name some jerk you know:");
    gets(jerk);
    puts("Yeah, I think");
    puts(jerk);
    puts("is a jerk, too.");
    return(0);
}
```

С помощью редактора модифицируйте программу INSULT3.C. Сохраните изменения на диске под именем INSULT4.C. Скомпилируйте и выполните.

```
Name some jerk you know:
David
Yeah, I think
David is a jerk, too.
```

Вывод выглядит несколько странно, примерно как в самых первых программах электронной почты. Но программа работает так, как было запланировано.



- ✓ Лучше не заменять `printf()` на `puts()`, а заново продумать стратегию вашей программы. Функция `puts()` автоматически ставит символ новой строки в конце отображаемой строки. Больше не нужно заканчивать строки символом `\n`! Кроме того, `puts()` может отобразить только одну строковую переменную за один вызов, притом на отдельной, ее собственной строке.



- ✓ Сначала в программе с помощью ключевого слова `char` нужно “объявить” строковую переменную. Затем нужно запомнить что-нибудь в этой переменной — это можно сделать с помощью функции `scanf()` или `gets()`. Только после этого будет иметь смысл отображать содержимое данной переменной с помощью `puts()`.



- ✓ Не применяйте `puts()` к переменной, если ее содержимое не является строкой. Вывод будет чем-то сверхъестественным. (См. главу 8 “Переменные в языке C”.)

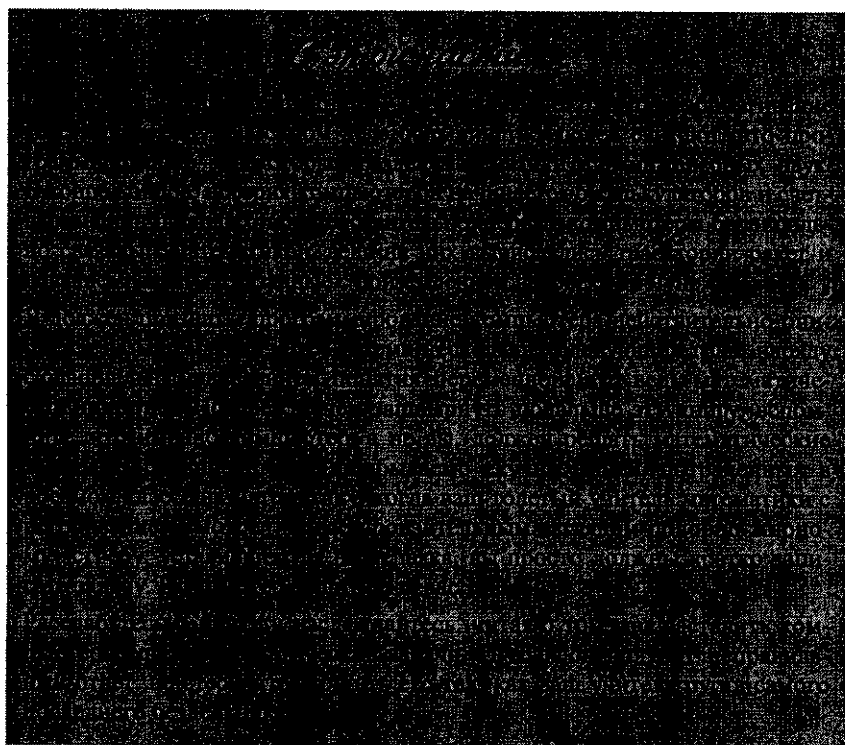
Когда использовать `puts()`, а когда `printf()`

- ✓ Используйте `puts()`, чтобы отобразить одну строку текста — ничего замысловатого.
- ✓ Используйте `puts()`, чтобы отобразить содержимое строковой переменной в отдельной строке.
- ✓ Используйте `printf()`, чтобы отобразить содержимое переменной, вставляемое в середину другой строки.
- ✓ Используйте `printf()`, чтобы с помощью одной инструкции отобразить содержимое нескольких переменных.
- ✓ Используйте `printf()`, если символ новой строки (тот, который получается при нажатии <Enter>) должен быть отображен после каждой строки, например, при вводе.
- ✓ Используйте `printf()`, когда требуется форматировать вывод.

Часть II

Переменные и некоторая доля математики





Глава 7

A + B = C

В этой главе...

- Изменение значения переменной
- Знакомство с `int`
- Преобразование текста с помощью функции `atoi()`
- Знаки `+`, `-`, `*` и `/`
- Основы математики

Пришло время подтвердить ваши самые мрачные опасения. Да, компьютеры имеют некоторое отношение к математике. Но это больше похоже на мимолетное увлечение, чем на безумную манию, от которой у вас трясутся поджилки. Если только вы не заядлый инженер, с маниакальным рвением проектирующий аппараты для потусторонней телепортации, математика играет случайную роль в ваших программах. Вы добавляете, вычитаете, делите, умножаете и, возможно, выполняете некоторые другие операции. Ничего такого, с чем не мог бы справиться калькулятор. Это действительно материал, который проходят в четвертом классе, но так как мы работаем с переменными, он более похож на материал уроков алгебры в восьмом классе — иногда придется немного пошевелить мозгами. В этой главе я попробую сделать это настолько приятным, насколько это возможно вообще.

Вечно изменяющаяся переменная

Переменная — место в памяти. Компилятор с языка C находит место в памяти, размещает и резервирует участок памяти, чтобы хранить в нем текстовые строки или иные значения — в зависимости от типа созданного места в памяти. Вам для этого нужно лишь поверхностное знание ключевых слов языка C, с которыми вы скоро познакомитесь.

Что можно хранить в памяти? Что угодно. Именно поэтому место в памяти называется переменной. Его содержимое может зависеть от того, что печатается на клавиатуре, от результата некоторой математической операции, предвыборного обещания или психоаналитического предсказания. Содержимое может измениться — точно так же, как психическое предсказание или предвыборное обещание.

Именно манипулирование переменными и составляет основную работу большинства программ. Когда вы запускаете, например, `РасМан`¹, позиция окна на экране сохраняется в переменной, потому что, в конце концов, оно перемещается (его позиция изменяется). Количество точек, отводимых для `РасМан`, также хранится в переменной. А когда вы выигрываете игру, вы вводите ваше имя, и оно также хранится в переменной. Значения этих элементов — местоположение окна `РасМан`, ваши очки, ваше имя — все это изменяется или может измениться, и поэтому хранится в переменных.

¹ Игра. — Прим. ред.



- ✓ Переменные — места в памяти для хранения информации, нужной программе. Они могут содержать числа, строки текста и другие элементы.
- ✓ Содержимое переменной может изменяться. Переменные — это то, что может хранить строки текста или числа. Их содержимое зависит от того, что происходит при выполнении программы, от того, что печатает пользователь, или от настроек компьютера, например. Переменные могут изменяться.
- ✓ Где хранятся переменные? В памяти компьютера. Компьютер отыскивает место для переменных, если вы следуете надлежащим процедурам создания переменных в программах на С.

Изменение (замена) строк

В следующей программе вы встретите ключевое слово `char` и функции `printf()` и `gets()`. В этой программе создана строковая переменная `kitty` (котенок); она используется дважды, когда пользователь решает, как назвать своего кота. Изменение содержимого `kitty` показывает природу переменной:

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

```
int main()                                // главная функция
{
    char kitty[20];
    // Как бы Вы хотели назвать вашего кота?
    printf("What would you like to name your cat?");
    gets(kitty);                          // котенок
    // хорошее имя. Что еще Вы имеете в виду?
    printf("%s is a nice name. What else do you have in mind?", kitty);
    gets(kitty);                          // котенок
    printf("%s is nice, too.\n", kitty);   // также хорошо
    return(0);
}
```

С помощью текстового редактора введите исходный текст `KITTY.C`. Сохраните файл на диске под именем `KITTY.C`.

Скомпилируйте `KITTY.C`. Если получите какие-нибудь сообщения об ошибках, отредактируйте заново исходный текст. Проверьте, не отсутствуют ли точки с запятой, на месте ли запятые и так далее. Затем перетранслируйте.

Выполнение программы описано в следующем разделе.

- ✓ Ключевое слово `char` помогает создать переменную и отводит память для ее хранения.
- ✓ Содержимое переменной может читаться только после присваивания ей текста.
- ✓ Именно функция `gets()` читает текст с клавиатуры и сохраняет его в строковой переменной.

Выполнение программы KITTY (КОТЕНОК)

После компиляции исходного текста `KITTY.C`, представленного в предыдущем разделе, выполните полученную программу. Вывод будет выглядеть примерно так:

```
What would you like to name your cat? Rufus
Rufus is a nice name. What else do you have in mind? Fuzzball
Fuzzball is nice, too.
```

(Как бы вы хотели называть вашего кота? Рыжик
Рыжик — хорошее имя. Что еще вы имеете в виду? Пухнастик
Пухнастик также хорошее имя.)

Переменной `kitty` (котенок) одно значение присваивается в первом вызове функции `gets()`. Затем ей присваивается новое значение во втором вызове функции `gets()`. Хотя используется одна и та же переменная, ее значение изменяется. Это — пример использования переменных.

- ✓ Одна переменная может использоваться в программе много раз независимо от того, имеет ли она то же самое значение или много раз меняет его.
- ✓ Отображается именно содержимое строковой переменной, а не имя переменной. В программе `KITTY.C` переменную называют `kitty` (котенок). Это название важно только для программиста. Для пользователя важно лишь то, что хранится в переменной.

Добро пожаловать в Холодный Мир Числовых Переменных

Точно так же, как строки текста хранятся в строковых переменных, числа хранятся в числовых переменных. Это позволяет программе работать со значениями и решать самые сложные математические задачи.

Чтобы создать числовую переменную и отвести память для числа, в языке C предусмотрены специальные ключевые слова. В отличие от ключевого слова `char`, которое создает все типы строк, для создания переменных, хранящих различные типы чисел, используются различные ключевые слова. Все зависит от величины числа и от того, как оно представлено.

Привет, целое число

Чтобы вы не запутались, пока я знакомлю вас только с одним типом числовых переменных. Это самый простой тип числа — целое число. Целое число по-английски — *integer*.

Вот как типичный компилятор с языка C определяет, что число относится к целочисленному типу:

- ✓ Целое число является целым, и потому в нем не может быть никаких дробей, десятичной части или еще какой-нибудь забавной штуковины.
- ✓ Неотрицательное целое число может иметь значение, которое находится в диапазоне от 0 до 32 767.

Неположительные числа находятся в диапазоне от -32 768 до 0.

Любые другие значения — большие или меньшие, дроби, или значения с десятичной запятой, типа 1,5 — это *не целые числа*. (Язык C может выполнять действия над такими числами, но пока я не ввожу такие типы переменных.)

Чтобы использовать целую переменную в программе, для нее нужно отвести память. Это делает ключевое слово `int`, которое ставится недалеко от начала программы. Вот — формат:

`int переменная величина;`

Ключевое слово `int` отделяется пробелом (или символом табуляции — при нажатии клавиши табуляции), а затем следует название (имя) переменной *переменная величина*. Это — законченная инструкция на языке C, и потому она заканчивается точкой с запятой.

- ✓ В некоторых компиляторах диапазон для `int` намного шире, чем от $-32\,768$ до $32\,767$. Чтобы выяснить это, загляните в документацию к вашему компилятору или в справочную систему.
- ✓ В старых, 16-разрядных компьютерах целое число должно было находиться в диапазоне от $-32\,768$ до $32\,767$.
- ✓ В самых современных компьютерах целые числа занимают диапазон от $-2\,147\,483\,647$ до $2\,147\,483\,647$.
- ✓ Подробнее об именовании переменных и других мелочах языка C, связанных с переменными, рассказывается в главе 8 “Переменные в языке C”. Пока же я ограничусь этим неофициальным введением.
- ✓ Да! Вы очень наблюдательны. Этот тип `int` — тот же самый, что обычно в каждой программе используется в объявлении главной функции `main()`. Не забегая слишком далеко вперед, я сообщу вам, что главная функция `main()` является “целочисленной функцией”. Она часто возвращает значение 0 в последней инструкции, но об этом вы узнаете подробнее в нескольких последующих главах.
- ✓ Компьютерные гуру всего мира хотят, чтобы вы знали, что целое число располагается в диапазоне от $-32\,768$ до $32\,767$ только на персональных компьютерах. Если вы случайно, от нечего делать когда-либо писали программы для какого-нибудь большого, старинного компьютера, то знаете, что диапазон для целых чисел на тех компьютерах несколько другой. Согласен, нет ни малейшей надобности забивать этой информацией вашу голову, но вся эта свора гуру скулила бы и истерически визжала, если бы могла заподозрить, что вы не знаете этого.



Использование целой переменной в программе Methuselah

Если в программе на языке C нужно обрабатывать только маленькие целые числа, следует использовать целые переменные. В качестве примера в следующей программе используется переменная `age` (возраст), чтобы хранить чей-то возраст. Другие примеры использования целых переменных: для хранения числа некоторых событий (если эти события происходят не больше чем $32\,000$ раз), для хранения количества планет в солнечной системе (все еще 9), количества коррумпированных конгрессменов (всегда меньше чем 524) и количества людей, которые лично видели Ленина (уменьшается с каждым днем). В общем, целые переменные пригодны для хранения целых чисел, но не больших.

Следующая программа отображает возраст Библейского патриарха Мафусаила, предка Ноя, который, возможно, дожил до 969 лет — достаточно долго, чтобы в этом усомниться. Программа называется `METHUS1.C`²:

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

```
int main()
```

```
// главная функция
```

² *Methus (Метюзл) — английское написание имени Мафусаил. Впрочем, в переводах Библии используются другие написания этого имени. В Синодальном издании, например, сказано: “Всех же дней Мафусала было девятьсот шестьдесят девять лет...” (Бытие, Глава 5, стих 27).* — Прим. ред.

```

{
    int age;                                // возраст

    age=969;                                // Мафусаилу было %d лет
    printf("Methuselah was %d years old.\n", age);
    return(0);
}

```

Введите текст METHUS1.C с помощью вашего редактора. Сохраните файл на диске под именем METHUS1.C.

Скомпилируйте программу. Если будут сообщения об ошибках, снова отредактируйте исходный текст и удостоверьтесь, что все соответствует предыдущей распечатке. Перетранслируйте.

Выполните программу, и вы увидите следующее:

Methuselah was 969 years old.

(Мафусаилу было 969 лет.)

Переменная `age` получила значение 969. Затем с помощью инструкции `printf()`, в которой используется заполнитель `%d`, это значение отображается в строке.

- ✓ Строка `int age;` создает переменную `age`, в которой хранится целочисленное значение.
- ✓ Строка `age=969;` присваивает значение 969 переменной `age` с помощью знака равенства (=). Сначала идет переменная `age`, затем знак равенства (=), и, наконец, само значение (969), которое должно быть помещено в переменную `age`.
- ✓ В строке `printf("Methuselah was %d years old.\n", age);` с помощью функции `printf` отображается значение переменной `age`. Для форматирования строки — параметра функции `printf()` — символ преобразования `%d` используется как заполнитель для целочисленного значения. Заполнитель `%d` используется для целых чисел так же, как заполнитель `%s` — для строк.

Присваивание значений числовым переменным

Читая программу METHUS1, заметьте, что числовым переменным значения присваиваются с помощью знака равенства (=). Переменная стоит слева от него, затем следует сам знак "=", а затем (справа от него) то, что дает значение. Вот как это записывается на языке C:

```
var=value;
```

Здесь *var* (от *variable* — переменная величина) — название (имя) числовой переменной; *value* (значение) — значение, которое присваивается этой переменной. Читайте эту строку так: "значение переменной *var* равно значению *value*."

Чем может быть *value* (значение)? Оно может быть числом, математическим выражением, функцией языка C, которая генерирует значение, или другой переменной, и в этом случае переменная величина *var* получит то же самое значение, что и данная переменная. В данном случае справа от знака равенства является приемлемым все, что вырабатывает значение — целочисленное значение, конечно.

Значение переменной `age` в METHUS1.C присваивается явно:

```
age=969;
```

Значение 969 можно сохранить в переменной `age`.



- ✓ Знак "=" позволяет присвоить переменной значение, не являющееся строкой. Переменная стоит слева от знака "=" и получает свое значение от того, что стоит справа от него.



✓ Строковые переменные не могут получить значения этим способом, т.е. с помощью знака “=”. Вы не можете сказать

```
kitty="Koshka";
```

Это просто не сработает! Строки могут читаться в переменные с клавиатуры с помощью `scanf()`, `gets()` или других функций чтения клавиатуры, определенных в языке C. Строковые переменные могут также предварительно получить значения, но не с помощью такого использования знака “=”, как с числовыми переменными!

Ввод числовых значений с клавиатуры

Рассмотрите программу `METHUS1.C` с помощью вашего редактора в течение нескольких секунд. Что она в действительности делает? Ничего. Поскольку значение 969 находится уже в программе, в ней нет ничего удивительного, и тем более, не ждите от нее какой-либо неожиданности. Реальная забава с числами происходит тогда, когда они вводятся с клавиатуры. Кто знает, какое дурацкое значение пользователь может ввести? (Именно поэтому и используются переменные.)

При чтении значения с клавиатуры возникает маленькая проблема: с клавиатуры читаются только строки; функции `scanf()` и `gets()`, с которыми вы знакомы, используются для чтения строковых переменных. И есть вполне определенное различие между символами “969” и числом 969. 969 — значение, а “969” — строка. Нужно тайно преобразовать строку “969” в значение — точнее, целочисленное значение — 969. Секретная команда, которая делает это — функция `atoi`.

Функция `atoi()`

Функция `atoi()` (произносится как “A-to-I”) конвертирует (преобразовывает) числа в начале строки в целочисленное значение. A — это первая буква акронима ASCII, схемы кодирования, которая присваивает секретные числа (номера) символам, т.е. кодирует их. Так что `atoi` значит “конвертировать (преобразовывать) текстовую (ASCII) строку в целочисленное значение”. Ниже показано, как читать целые числа с клавиатуры. Вот формат инструкции:

```
var=atoi(string); // переменная=atoi(строка);
```

Здесь `var` (переменная величина) — название (имя) числовой переменной, притом целочисленной переменной, т.е. созданной ключевым словом `int`. После него следует знак “=”, который используется для присваивания значения переменной.

Функция `atoi()` следует за знаком “=”. Затем в круглых скобках `atoi()` следует строка, которую нужно конвертировать (преобразовать). Строка может быть строковой переменной или строковой “константой”, заключенной в двойные кавычки. Наиболее часто строка `string`, которую нужно конвертировать (преобразовать), представляет собой название (имя) строковой переменной, созданной ключевым словом `char`, причем сама строка читается с клавиатуры с помощью функции `gets()`, `scanf()` или некоторой другой функции чтения клавиатуры.

В конце строки ставится точка с запятой, потому что это законченная инструкция языка C.

Функция `atoi` также требует еще одной директивы включения в начале исходного текста:

```
#include <stdlib.h>
```

Эта строка обычно помещается ниже традиционной директивы `#include <stdio.h>` — они обе фактически выглядят одинаково, но в угловых скобках теперь — `stdlib.h`, именно эта библиотека требуется для `atoi`. Строка не заканчивается точкой с запятой.

- ✓ Имя `atoi` читается как “А-то-І”, а не как “a toy” (“игрушка”).
- ✓ Числа — это значения; строки составлены из символов.
- ✓ Если строка, которую конвертирует `atoi()`, начинается не с числа, или если число является слишком большим или записано в неестественном формате для целого числа, `atoi` возвращает значение 0 (нуль).
- ✓ Назначение `#include <stdlib.h>` состоит в том, чтобы сообщить компилятору о функции `atoi()`. Если этой строки нет, компилятор может сгенерировать несколько предупреждений или ошибки типа “no prototype” (“нет прототипа”), которые обычно портят вам настроение на весь день.
- ✓ `STDLIB.H` — стандартный библиотечный файл заголовка.
- ✓ Для преобразования (конвертирования) строки в целое число в языке С предусмотрены другие функции. Вот как ввод с клавиатуры преобразуется в числовое значение: строка помещается внутрь скобок специальной функции (`atoi`), которая возвращает число.



О различии между числами и строками, если вам это все еще интересно

Вы должны определить, когда число в С — значение, а когда оно — строка. Числовое значение запоминается в числовой переменной. В этой книге такие вещи чаще называются **значениями**, а не числами. Значение — 5 яблок, 3,141 (например), национальный долг, количество килограммов, которые вы можете потерять, если будете соблюдать диету какой-нибудь звезды, сумевшей добиться знаменитости на этой неделе. Это значения.

Числа — это то, что появляется в строках текста. Когда вы печатаете 255, например, вы вводите строку. Это — символы 2, 5 и 5, которые находятся на вашей клавиатуре. Строка “255” — не значение. Я называю ее числом³. Функция `atoi()` языка С преобразовывает число в значение, подходящее для хранения (запоминания) в числовой переменной.

Есть числа, а есть значения. Как их отличить? Это зависит от того, как вы собираетесь использовать их. Очевидно, если кто-то вводит номер телефона, номер дома или почтовый индекс, это, по всей вероятности, строка. (Мой почтовый индекс — 94402, но отсюда не следует, что это 94 402-е почтовое отделение в Соединенных Штатах.) Если кто-то вводит количество долларов, процент, размер или результат какого-нибудь измерения, — что угодно, над чем можно выполнять математические действия, — это, вероятнее всего, значение.

Так сколько же лет было Мафусаилу?

Следующая программа — `METHUS2.C` — очередное незначительное, но верное усовершенствование `METHUS1.C`. В этой версии программа читает строку, печатаемую пользователем на клавиатуре. Эта строка, а это и в самом деле строка, затем волшебным образом преобразовывается в числовое значение функцией `atoi()`. После этого полученное значение отображается функцией `printf()`. Именно в этом и состоит Чудо.

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

```
int main()
{
```

```
// главная функция
```

³ Иными словами, числом автор часто называет представление числа в виде строки. — Прим. ред.

```

int age; // возраст
char years[8];

printf("How old was Methuselah?"); // Сколько лет было Мафусаилу?
gets(years); // возраст
age=atoi(years);
// Мафусаилу было %d лет
printf("Methuselah was %d years old.\n",age);
return(0);
}

```

Загрузите исходный текст METHUS2.C в ваш редактор. Вы можете редактировать первоначальный исходный текст METHUS1.C, но будьте внимательны, и сохраните новый исходный текст на диске под новым названием (именем) METHUS2.C.

Скомпилируйте программу. Исправьте все ошибки.

Выполните программу. Вывод будет подобен следующему:

```

How old was Methuselah?26
Methuselah was 26 years old.

```

(Сколько лет было Мафусаилу? 26
Мафусаилу было 26 лет.)

В этом примере пользователь напечатал 26 в качестве возраста. Это число было введено как строка, преобразовано функцией `atoi()` в целочисленное значение и, наконец, отображено функцией `printf()`. Вот так можно читать числа с клавиатуры и затем в программе преобразовывать их в числовые значения. В других разделах этой главы, так же как и в остальной части этой книги, мы будем пользоваться только что изученным приемом.

- ✓ Легенда гласит, что старику было 969, когда он наконец (и вероятно счастливо) вступал в будущую жизнь. Но, используя эту программу, можно изменить эту хронологическую деталь (хотя у Мафусаила, вероятно, было много современников, которые жили столько же, сколько вы и я).
- ✓ Если вы забыли о директиве `#include <stdlib.h>`, это приведет к орфографическим ошибкам, причем компилятор может сгенерировать несколько ошибок. Обычно эти ошибки — неопасные “предупреждения”, и программа будет работать все равно. Но независимо от этого, сделайте своей привычкой включать библиотеку `stdlib`, когда используете функцию `atoi()`.
- ✓ Вот как функция `atoi` используется для преобразования строки `years` в числовое значение: `age=atoi(years);`. Функция `atoi()` исследует строку и возвращает число. Затем это число помещается в переменную `age` как числовое значение.
- ✓ Почему же мы не печатаем строку `years`? Да, это сделать можно. Замените `age` (и `%d` на `%s`) в заключительной функции `printf()`, и программа будет отображать то же самое сообщение. Строка должна выглядеть вот так:
`printf("Methuselah was %s years old.\n",years);`

Вывод не изменился. Однако только над числовой переменной можно выполнять математические операции. Строки и математика? Не морочьте голову, рассуждайте здраво!





Конечно, вы не обязаны экспериментировать с METHUS2.C, но я рекомендую вам попробовать это

Слава Богу, эта книга — не *Surgery For Dummies* (Хирургия для чайников), и потому с программами, приведенными в данной книге, можно играть в свое удовольствие, записывать их по какому-нибудь машинному адресу, перемещать, экспериментировать и забавляться.

Снова выполните программу METHUS2.C, но когда программа спросит у вас возраст Мафусаила, напечатайте следующее значение:

10000000000

Это десять миллиардов — единица с 10 нулями без запятой. Нажмите <Enter>, и программа скажет вам, что старичку было 1 410 065 408 лет — или выведет некоторое другое значение, а не то, что вы напечатали. Причина состоит в том, что вы ввели значение большее, чем наибольшее целое число, которое может храниться в целочисленной переменной. Возвращенное значение — остаток от деления введенного числа на максимальную величину целого числа, допустимую для вашего компилятора.

А что будет, если напечатать следующее значение:

-64

Да, г-ну М. никогда не могло быть -64 года, но программа не обращает внимания на это. Причина состоит в том, что отрицательные числа также относятся к целочисленным значениям.

Можете попробовать ввести еще одно число:

4.5

Самый старый человек — действительно ли ему могло быть четыре с половиной года? Вероятно, когда-то, да. Однако программа настаивает, что ему было только четыре. Дело в том, что точка, за которой следует цифра 5, — дробная часть. Целые числа не могут иметь дробных частей, поэтому функция `atoi()` считывает 4.

Наконец, поставим следующий большой эксперимент. Напечатайте следующее как возраст Мафусаила:

old

(старый)

Да, он был стар. Но когда Вы вводите `old` (старый) в программу, она утверждает, что ему было только нуль лет. Причина состоит в том, что функция `atoi()` не нашла числа в вашем ответе. Поэтому возвращает в качестве значения нуль.

А сколько лет вам?

Пришло время снова изменить старую программу METHUS. На сей раз создадим исходный текст METHUS3.C, как показано ниже. Как и в случае METHUS2.C, в первоначальную программу потребуются внести только незначительные модификации. Ничего нового.

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

```
int main() // главная функция
{
    int methus;
    int you;
    char years[8];

    printf("How old are you?"); // Сколько вам лет?
    gets(years); // годы
    you=atoi(years);
```

```

printf("How old was Methuselah?"); // Сколько лет было Мафусаилу?
gets(years);                       // годы
methus=atoi(years);

printf("You are %d years old.\n",you); // Вам %d лет
// Мафусаилу было %d лет
printf("Methuselah was %d years old.\n",methus);
return(0);
}

```

Тщательно перепроверьте исходный текст и сохраните файл на диске под именем METHUS3.C.

Скомпилируйте METHUS3.C. Устраните все вкравшиеся ошибки.

Выполните программу. Она задаст два вопроса, а затем отобразит две строки текста — получится что-то вроде следующего:

```

How old are you? 29
How old was Methuselah? 969
You are 29 years old.
Methuselah was 969 years old.

```

(Сколько вам лет? 29
Сколько было Мафусаилу? 969
Вам 29 лет.
Мафусаилу было 969 лет.)

Конечно, эти данные просто выведены без обработки. Истинная мощь компьютера заключена в тех математических действиях, которые он может выполнить над значениями, потому что именно компьютер решает математические задачи, а не вы. Не проклиняйте его!



- ✓ Вам на самом деле не 29 лет.
- ✓ Да, переменная-строка `years` используется дважды. Сначала в нее читается ваш возраст, а затем функция `atoi()` конвертирует (преобразовывает) его и сохраняет полученное значение в переменной `you`. Затем `years` используются снова для ввода. Этот подход работает, потому что первоначальное значение было сохранено в другой переменной — крутая уловка!
- ✓ Программа METHUS3.C была разделена на четыре раздела. Это больше относится к визуальной организации, чем к языку программирования C. Идея записывать программу по частям подобна способу, которым большинство людей пробует писать на английском языке.

Крошечный бит математики

Теперь пришло время всем хорошим программистам заняться математикой. Нет, подождите! Пожалуйста, не волнуйтесь. Это совсем простой материал. Первое реальное математическое объяснение еще далеко — до него еще несколько страниц.

Решая математические задачи с помощью языка программирования C, вы должны знать две вещи. Во-первых, вы должны знать знаки математических операций, т.е. знать, какие символы обозначают сложение, вычитание, умножение и деление чисел. Во-вторых, вы должны знать, что делать затем с результатами. Конечно, вы ни в коем случае не должны решать математические задачи. Ведь как раз для этого предназначен компьютер.

Основные математические символы

Основные математические символы, вероятно, знакомы вам. Вот они:

- ✓ символ сложения: +
- ✓ символ вычитания: -
- ✓ символ умножения: *
- ✓ символ деления: /

Кстати, официальный термин языка C для этих болванов — операторы. Они являются математическими (или арифметическими — я никогда не знаю, какое слово использовать) операторами.

+ Сложение: операция сложения обозначается знаком “плюс” +. Этот знак является настолько важным, что я действительно не могу придумать что-нибудь, где бы не понадобилось сложить два числа:

```
var=value1+value2;
```

Здесь результат добавления value1 к value2 вычислен компьютером и сохранен в числовой переменной var.

- Вычитание: оператор вычитания — знак “минус” -:

```
var=value1-value2;
```

Здесь результат вычитания value2 от value1 вычислен и сохранен в числовой переменной var.

*** Умножение:** для умножения используется непривычный знак. Оператор умножения — звездочка, а не символ X:

```
var=value1*value2;
```

В этой строке результат умножения value1 на value2 вычислен компьютером, а результат сохранен в переменной var.

/ Деление: для обозначения деления используется наклонная черта вправо /; главная причина состоит в том, что символа ÷ нет на клавиатуре:

```
var=value1/value2;
```

Здесь результат деления value1 на value2 вычислен компьютером и сохранен в переменной var.



Обратите внимание, что во всех случаях математическая операция находится справа от знака “=”. Следующая запись недопустима:

```
value1+value2=var;
```

Это не работает на языке C. Предыдущая строка велит компилятору с языка C взять значение переменной var и поместить его в некоторые числа. Как это? Это — вид ошибки, называемой ошибкой именуемого выражения (ошибка адреса переменной); этот вид ошибок позорно популярен.



- ✓ В языке C предусмотрены различные математические символы, или операторы. В этой главе вводится только четыре наиболее часто используемых символа.
- ✓ У вас были неприятности с математическими операторами? Посмотрите на цифровую клавиатуру (она справа на вашей клавиатуре)! Наклонная черта вправо, “звездочка”, “минус” и символ “плюс”, а также цифровые клавиши.



- ✓ В отличие от того, с чем мы встречаемся в реальном мире, вы должны твердо помнить, что вычисления всегда записываются справа, а ответ — слева от знака “=”.
- ✓ Вы не всегда должны работать с двумя явно выписанными значениями в математических функциях. Вы можете работать с переменной и значением, двумя переменными, функциями — здесь возможно много вариантов. Язык C гибок, но в этой стадии важно помнить только то, что знак умножения — звездочка *.



Почему звездочка является символом умножения — если вам хочется это знать

В школе вы, вероятно, узнали, что символ \times обозначает умножение. Точнее, это — символ \times , а не символ x (или даже строчная буква x). Он читается как “умножить”, но 4×9 можно читать и как “четыре раза по девять”. В высшей математике для обозначения умножения также используется точка: $4 \cdot 9$ читается как “четыре раза по девять”. Правда, когда я увидел, как некоторые пытались записать это в виде $4(9)$ или $(4)9$, я заснул.

Почему компьютеры не могут использовать \times ? Прежде всего потому, что они глупы. Компьютер не знает, когда вы подразумеваете под x букву “икс”, а когда \times (“умножить”). Поэтому в качестве заместителя этого знака была принята звездочка (*). (Ведь и подходящего символа вроде точки \cdot на клавиатуре тоже нет.)

Конечно, потребуется некоторое время, чтобы привыкнуть к использованию * для обозначения умножения. Наклонная черта вправо более привычна — $3/\$1$ обозначает “три за доллар” или 33 цента каждая единица товара, — так что привыкнуть к наклонной черте — не проблема. Но вот со звездочкой * сложнее.

Сколько еще вы должны прожить, чтобы побить рекорд Мафусаила?

Следующий исходный текст программы METHUS4.C использует немного математики, чтобы выяснить, сколько лет еще вы должны жить, чтобы побить рекорд (или хотя бы сравняться с ним) самого старого человека, жившего когда-либо.

Прежде чем вы сделаете это, я хочу, чтобы вы подумали об этом. Какие преимущества того, чтобы стать самым старым человеком, когда-либо жившим до этого? Что еще мы знаем о Мафусаиле? Он умер перед Потопом. Он был хорошим человеком и был хорошо принят Богом. Ну, что еще? Я хочу спросить, что он ел — нечто сверхъестественное или кое-что другое? Он избегал винограда? В Библии нет никаких подсказок.

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

```
#include <stdlib.h>
```

```
int main() // главная функция
{
    int diff; // разность
    int methus;
    int you;
    char years[8];

    printf("How old are you?"); // Сколько вам лет?
    gets(years); // годы
    you=atoi(years);

    methus=969; /* Мафусаилу было 969 лет */
```

```
diff=methus-you;
```

```
    // Вы на %d лет моложе Мафусаила
printf("You are %d years younger than Methuselah.\n",diff);
return(0);
}
```

Программа METHUS4.C устрашающе подобна METHUS3.C. В ней только несколько отклонений от METHUS3.C, так что можете отредактировать исходный текст METHUS3.C и затем сохранить его на диске под именем METHUS4.C. Сверьте вашу работу с предыдущей распечаткой исходного текста, только чтобы убедиться, что все в порядке.

Скомпилируйте программу. Исправьте все ошибки. Затем выполните конечный программный продукт. Вывод будет подобен следующему:

```
How old are you?29
```

```
You are 940 years younger than Methuselah.
```

```
(Сколько вам лет? 29
```

```
Вы на 940 лет моложе Мафусаила.)
```

Попробуйте ввести различные значения возраста, чтобы увидеть, как программа сравнивает эти значения с возрастом Мафусаила.

- ✓ Компьютер решает математические задачи! Компьютер решает математические задачи!
- ✓ Эта программа использует четыре — четыре! — переменные: diff, methus и you — все целочисленные переменные, а years — строковая переменная.
- ✓ Что случится, если ввести отрицательный возраст или возраст, превосходящий 969?
- ✓ Чтобы выяснить, сколько еще вам нужно прожить, чтобы побить рекорд Мафусаила, вы вычитаете ваш возраст из его возраста. Обратите особое внимание на порядок переменных в следующей инструкции:

```
diff=methus-you;
```

Математические действия всегда указываются справа от знака “=”. Ваш возраст, хранимый в числовой переменной, вычитается из возраста Мафусаила и хранится в числовой переменной diff. Результат как бы проскальзывает через знак “=” в числовую переменную diff.

Дополнительная модификация заключительной версии программы METHUS

Мафусаил интенсивно работал до 65 лет, а затем он “вышел на пенсию”. Поскольку он впервые получил работу в 19 лет, он вносил страховые отчисления в фонд Социального обеспечения. Как только он достиг 65, он начал получать деньги. И так все время. И еще. И потом еще.

Как написать программу, которая сможет сделать то, что не может сделать ни один бюрократ в Вашингтоне: выяснить, сколько денег Мафусаил вытянул из фонда? (Мы отложим решение вопроса “заработал ли он все эти деньги” для будущих поколений, потому что, конечно, если вы спросите самого Мафусаила, он скажет, что он таки заработал.)

```
#include <stdlib.h>
```

```
int main()
```

```
{
    int contributed;
    int received;
    contributed=65-19;
```

```
// главная функция
```

```
// пожертвования
```

```
// полученные деньги
```

```
received=969-65;
```

```
// Мафусаил вносил вклад в фонд Социального обеспечения в течение %i лет
printf("Methuselah contributed to Social Security for %i years.\n",
      contributed);
// Мафусаил получал из фонда Социального обеспечения в течение %i лет
printf("Methuselah collected from Social Security for %i years.\n",
      received);
return(0);
}
```

Введите с помощью редактора исходный текст METHUS5.C, который, — я это твердо обещаю, — последний в наборе программ о Мафусаиле. Перепроверьте введенный текст и исправьте его в случае необходимости. (Возможно, вы напечатали *i* перед *e* в *received*, но нелогичность английского языка требует изменить порядок этих букв.)

Сохраните под именем METHUS5.C.

Скомпилируйте программу! Исправьте все ошибки, даже самые маленькие, чтобы компилятор не нашел ни одной. Для этого заново отредактируйте исходный текст и скомпилируйте его снова, если нужно.

Выполните программу! Вот пример вывода:

```
Methuselah contributed to Social Security for 46 years.
Methuselah collected from Social Security for 904 years.
```

(Мафусаил вносил в фонд Социального обеспечения в течение 46 лет.

Мафусаил получал из фонда Социального обеспечения в течение 904 лет.)

- ✓ Ему это кажется справедливым.
- ✓ Оператор `received=969-65;` вычисляет, как долго Мафусаил получал из фонда Социального обеспечения. Если он умер в 969 лет и начал получать пенсию в 65, то разность и есть искомое значение. Оно хранится в переменной `received`.
- ✓ Заметьте, что меньшее значение вычитается из большего. Математические действия в C выполняются слева направо: 65 минус 19; 969 минус 65. Однако математические действия записываются в правой части равенства. Переменная, которая хранит результат, указывается слева.
- ✓ Если математические действия в программе выполняются над числами, а не над переменными, эти числа называют константами.
- ✓ Более подробно о константах рассказывается в главе 8 “Переменные в языке C”.

Прямой результат

Действительно ли необходимы переменные? Да, когда значение неизвестно. В нескольких последних программах METHUS большинство значений было известно. В действительности переменная нужна только тогда, когда вы вводите ваш собственный возраст. Во всех остальных случаях могут использоваться постоянные значения.

Например, следующая программа представляет собой еще одну версию METHUS5. Вы не обязаны вводить эту программу, но посмотрите на две инструкции программы (кроме них там других инструкций нет). Нет переменных и инструкций, которые присваивали им значения, нет даже `#include <stdlib.h>`, потому что `atoi()` не используется:

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

```
int main()
```

```
// главная функция
```



```

{
// Мафусаил вносил в фонд Социального обеспечения в течение %d лет
printf("Methuselah contributed to Social Security for %d years.\n"
      ,65-19);
// Мафусаил получал из фонда Социального обеспечения в течение %d лет
printf("Methuselah collected from Social Security for %d years.\n"
      ,969-65);
return(0);
}

```

В первой функции printf() целочисленное значение вставляется вместо %d. Функция printf() ищет это значение после запятой — и оно там есть! Значение вычислено компилятором с языка C как 65–19, что равно 46. Инструкция printf() вставляет значение 46 вместо заполнителя %d. То же самое справедливо и для второй функции printf().

Все это вы можете сделать и без математики. Вы можете сами вычислить 65–19 и 969–65 и затем вставить эти значения непосредственно в программу:

```

// Мафусаил вносил вклад в Социальное обеспечение в течение %d лет
printf("Methuselah contributed to Social Security for %d years.\n",46);
// Мафусаил получал из фонда Социального обеспечения в течение %d лет
printf("Methuselah collected from Social Security for %d years.\n",904);

```

И снова результат будет тот же самый. %d ищет целочисленное значение, находит его и вставляет в отображаемую строку. Для printf() ведь не имеет значения, является ли это значение константой, математической формулой или переменной. Но значение должно быть обязательно целочисленным.

Переменные в языке C

В этой главе...

- Объявление переменных
- Обозначение (именование) переменных
- Использование переменных с плавающей точкой
- Объявление нескольких переменных сразу
- Суть констант
- Создание констант с помощью `#define`
- Использование ключевого слова `const`

Именно благодаря переменным выполнение программы изменяется от прогона к прогону. В программировании без переменных обойтись никак нельзя. Вы уже использовали переменные, но формально они вам еще не представлены. В этой главе как раз и состоится формальное представление!

Обсуждение, именование и объявление переменных

Итак, к нам прибывает Валери Вариабл (Valerie Variable)...

Валери — числовая переменная. Она любит хранить числа — любые числа — не имеет значения, какие. Всякий раз, когда она видит знак "=", она берется за значение и крепко обнимает его в своих объятиях. Но как только она видит другой знак "=", она принимает новое значение. В общем, Валери немного изменчива и переменчива. Вы можете сказать, что значения Валери изменяются, но ведь именно поэтому она и является переменной.

Ее очередной победитель — Виктор (Victor) — строковая переменная. Он хранит текст — все, что состоит из символов (в строке может быть от одного символа до нескольких). Виктору важно лишь, чтобы то, что он хранит, состояло из символов. Но вот какие именно символы, Виктору все равно, потому что он — тоже переменная, и он может хранить любые символы.

- ✓ В языке C имеется два основных типа переменных: числовые переменные, которые содержат только числа или значения, и строковые переменные, которые содержат текст, состоящий из символов (от одного до нескольких).
- ✓ Язык C имеет несколько различных типов числовых переменных в зависимости от размера и точности числа. Подробнее об этом рассказывается в главе 9 "Числа в языке C".
- ✓ Перед использованием переменную нужно объявить. Объявление — это..., Впрочем, лучше читайте следующий раздел.



Зачем объявлять переменные

Перед использованием переменных вы обязаны объявить их, чтобы компилятор языка C был осведомлен о них. Чтобы объявить переменные, укажите список переменных недалеко от начала исходного текста. Благодаря этому компилятор будет знать имена переменных, а также переменными какой разновидности они являются (а, значит, и какие значения они могут содержать). Официально этот процесс называется объявлением переменных.

Например:

```
int count;
char key;
char lastname[30];
```

В этом примере объявлены три переменные: целая переменная `count`; символьная переменная `key` и символьная переменная `lastname` (она является строкой, которая может содержать до 30 символов).

Объявление переменных в начале программы сообщает компилятору вот что. Во-первых, оно сообщает, что перечисленные в нем имена — имена переменных! Поэтому когда компилятор видит `lastname` в программе, он знает, что это — строковая переменная.

Во-вторых, в объявлениях компилятору сообщается тип переменной. Например, компилятор знает, что целочисленные значения записываются в переменную `count`.

В-третьих, компилятор знает, сколько места в памяти нужно отвести для переменных. Это не может быть сделано “на лету”, когда программа уже выполняется. Место должно быть отведено тогда, когда программа создается компилятором.



- ✓ Объявляют переменные в начале программы, сразу после строки с начальной фигурной скобкой. Соберите их все тут же.
- ✓ Очевидно, вы не можете перечислить все переменные, которые потребуются в программе прежде, чем вы запишете ее. (Хотя этому учат в университетах, это не требуется ни вам, ни мне.) Если понадобится новая переменная, с помощью редактора объявите ее в программе. Жуликоватые переменные — необъявленные — генерируют синтаксические ошибки или ошибки при компоновке (в зависимости от того, как используются эти переменные).
- ✓ Если какая-то переменная не объявлена, программа не компилируется. Будет сгенерировано сообщение об ошибках.
- ✓ В программе на C большинство программистов помещает пустую строку между объявлениями переменных и остальной частью программы.
- ✓ Можете прокомментировать, что содержит переменная. Например:

```
int count; /* счетчик сигналов занятости
            от службы технической поддержки. */
```
- ✓ Однако если название переменной говорит само за себя, можете опустить комментарий:

```
int busysignals;
```
- ✓ Впрочем, лучше назвать переменную вот так:

```
int busy_signal_count;
```

Запрещенные и допустимые имена переменных

Строго говоря, именование переменных зависит от компилятора. Необходимо следовать нескольким правилам, причем некоторые названия (имена) нельзя использовать для переменных. Если вы нарушите эти правила, компилятор сгенерирует сообщение об ошибках. Чтобы избежать этого, придерживайтесь следующих рекомендаций при создании новых переменных:

- ✓ Самое короткое имя переменной --- буква (символ алфавита).
- ✓ Имена используемых переменных должны быть осмысленными. Однобуквенные переменные удобны только для набора. Но `index` (индекс) лучше, чем `i`, `count` (счетчик) лучше, чем `c`, а `name` (название, имя) — лучше, чем `n`. Наилучшими являются короткие, описательные имена переменных.
- ✓ Имена переменных обычно печатаются на нижнем регистре. (В C предпочтение главным образом отдается нижнему регистру.) Они могут содержать буквы и цифры.
- ✓ Прописные буквы также могут использоваться в именах переменных, но большинство компиляторов имеет тенденцию игнорировать различия между символами верхнего и нижнего регистра. (Некоторые компиляторы могут учитывать регистр, для этого нужно установить одну из опций такого компилятора; подробности должны быть освещены в интерактивной справочной системе компилятора.)
- ✓ Имя переменной не может начинаться с цифры. Имя может содержать цифры, но начинается оно с буквы. Даже если ваш компилятор не генерирует сообщение об ошибке, если имя начинается не с буквы, другие компиляторы языка C так не думают, так что вы должны начинать имя переменной с буквы.
- ✓ Знатоки языка C используют подчеркивание (точнее, символ подчеркивания) в именах переменных: они, например, могут написать `first_name` и `zip_code`. Эта методика эффективна, хотя не рекомендуется начинать имя переменной с подчеркивания.
- ✓ Названия переменных не должны совпадать с ключевыми словами языка C или именами функций. Не называйте целую переменную таким именем, как, например, `int`, а строковую переменную — таким именем, как, например, `char` (символ). Компилятор может и не сгенерировать ошибку, но это запутывает исходный текст. (Обратитесь к табл. 3.1 из главы 3 “Формальное знакомство с языком C”, чтобы восстановить в памяти список ключевых слов языка C.)
- ✓ Избегайте также использовать одиночные символы `l` (нижний регистр для *L*) и `o` (нижний регистр для *O*) в качестве названий переменных. Строчный эквивалент `L` слишком уж похож на `1` (один), а строчный эквивалент `O` похож на `0` (ноль).
- ✓ Не давайте похожих названий (имен) разным переменным. Например, компилятор может предположить, что `forgiveme` и `forgivemenot` — та же самая переменная. В этом случае может произойти непредвиденная ситуация.
- ✓ Где-нибудь в загадочных файлах справки вашего компилятора скрыты официальные правила именования переменных. Эти правила свои у каждого компилятора, поэтому я не могу перечислить их все здесь. В конце концов, мне платят не по часам. И это — не составная часть моего контракта.



Предварительная установка значений переменных

Предположим, что парню по имени Мафусаил 969 лет. Я понимаю, что это может быть трудно принять, но давайте порассуждаем вместе.

Если вы хотите использовать возраст Мафусаила как значение в программе, вы можете создать переменную `methus` и затем запихнуть в нее значение 969. Это требует двух шагов. Сначала идет объявление:

```
int methus;
```

Эта строка говорит компилятору, что в `methus` можно хранить целочисленные значения. Затем следует присвоение, и 969 помещается в переменную `methus`:

```
methus=969;
```

В языке C можно объединить оба шага в один. Например:

```
int methus=969;
```

Эта инструкция создает целочисленную переменную `methus` и присваивает ей значение 969 — все делается в одной инструкции. Это ваше первое использование одного из многочисленных сокращений, предусмотренных в языке C. (Скажу прямо, что сокращений и альтернатив в C столько, что можно свихнуться.)

То же самое можно делать и со строковыми переменными, но в этом случае трюк будет выглядеть немного необычно. В большинстве случаев при создании строковых переменных указывается их размер. Например:

```
char prompt[22];
```

Здесь создана строковая (символьная) переменная `prompt` (подсказка), и ей отведен участок памяти для 22 символов. После этого `gets()` или `scanf()` может занести текст в эту переменную. (В этом случае не используется знак “=”!). Но если переменная создается и ей присваивается строка в одной инструкции, то формат этой инструкции немного иной. Вот пример:

```
char prompt[] = "So how fat are you, anyway?";
```

Данная команда создает строковую переменную `prompt` (подсказка). Эта строковая переменная уже содержит текст “So how fat are you, anyway?” (“Так сколько же ты вешишь?”). Обратите внимание, что количество символов в скобках не указано. Причина состоит в том, что компилятор достаточно хитер, чтобы вычислить длину строки и потом использовать вычисленное значение автоматически. Никаких гаданий — какая радость!

✓ Числовым переменным можно присвоить значение в объявлении. Для этого после имени переменной поставьте знак “=”, а за ним — присваиваемое значение. Не забудьте в конце строки поставить точку с запятой.

✓ Значение, присваиваемое переменной, может быть результатом математических действий. Например:

```
int video=800*600;
```

Эта инструкция создает целочисленную переменную `video` (видео) и устанавливает ее значение равным числу 800, умноженному на 600, т.е. 480 000. (Помните, что в C для обозначения умножения используется *.)

✓ Даже если переменной присвоено значение, значение этой переменной все же может изменяться. Пусть, предположим, создана целочисленная переменная `methus` и ей присвоено значение 969. Тогда не будет ничего неправильного, если позже в программе изменить значение этой переменной. В конце концов, переменная — она и есть переменная!

✓ Вот еще один трюк. (Правда, запоминать его совсем не обязательно.)

```
// пуск = начало = сначала = счетчик = 0;  
int start = begin = first = count = 0;
```

Эта инструкция объявляет четыре целочисленные переменные: `start` (пуск), `begin` (начало), `first` (сначала) и `count` (счетчик). Каждая из них устанавливается равной 0. Значение `start` равняется `begin`, что в свою очередь равно `first`, что в свою очередь равно `count`, что равно 0.



Старая случайная программа с переменными

Чтобы продемонстрировать возможные определения переменных с присваиванием определенных значений, я придумал программу ICKYGU.C. Она работает подобно тем старым китайским забегаловкам типа “все это вы можете съесть”, где стеклянные колпаки защищают от чихания ароматную бурду на лотках, обильно намазанных жиром. Ах... эта программа напоминает мне о днях, проведенных в колледже, и тех кишечных инфекциях, которые мне пришлось перенести. Вот ее исходный текст:

```
#include <stdio.h>

int main()                                // главная функция
{
    char menuitem[] = "Slimey Orange Stuff \"Icky Woka Gu\"";
    int pints=1;
    float price = 1.45;

    // Сегодня специальное блюдо
    printf("Today special - %s\n",menuitem);
    // ("Вы хотите %d пинт \n", пинты)
    printf("You want %d pint.\n",pints);
    // ("Это будет $%f, пожалуйста \n ", цена)
    printf("That be $%f, please.\n",price);
    return(0);
}
```

Напечатайте этот исходный текст с помощью редактора. Перепроверьте все. Сохраните программу под именем ICKYGU.C.

Скомпилируйте программу. Исправьте все ошибки и перетранслируйте в случае необходимости.

Выполните полученную программу. Вы увидите нечто подобное следующему:

```
Today special - Slimey Orange Stuff "Icky Woka Gu"
You want 1 pint.
That be $1.450000, please.
```

(Сегодня специальное блюдо — нечто оранжевое неприглядное

Вы хотите 1 пинту.

Это будет 1,450000 \$, пожалуйста.)

Стоп! Это лиры или доллары? Конечно, это доллары, ведь знак доллара в строке, отформатированной printf(), — обычный символ, а не что-нибудь специальное. Но это значение 1.45 напечатано с четырьмя дополнительными нулями. Почему? Потому, что вы не сказали компилятору, что они не нужны. Так символ преобразования %f форматирует числа с плавающей точкой.

Чтобы вывод меньше пугал, а цена в долларах выглядела более привычно, отредактируйте строку That be \$%f, please.\n, заменив в ней заполнитель %f на %.2f:

```
// ("Это будет $%f, пожалуйста \n ", цена)
printf("That be $%.2f, please.\n",price);
```

Со вставкой дополнительных символов между % и f вы уже знакомы; я воспользовался этим трюком в предыдущих главах, чтобы ограничить область форматирования для заполнителя %s. Здесь же printf() форматирует число с плавающей запятой, оставляя только две позиции для цифр после десятичной точки.

Сохраните измененный файл на диске. Перетранслируйте и выполните. Вывод будет более привлекательным:

Today special - Slimey Orange Stuff "Icky Woka Gu"
You want 1 pint.
That be \$1.45, please.

(Сегодня специальное блюдо — нечто оранжевое неприглядное
Вы хотите 1 пинту.
Это будет 1,45 \$, пожалуйста.)



- ✓ Эта программа содержит три типа переменных: строку `menuitem`; целочисленное значение `pints` (пинты) и значение с плавающей точкой `price` (цена).
- ✓ `price` (цена) — значение с плавающей точкой, потому что оно имеет десятичную точку. Это еще один тип числовой переменной. В отличие от целого числа, значения с плавающей точкой могут содержать десятичную точку.
- ✓ “Плавающая точка (запятая)” — это точка (запятая) в середине числа (такого, как, например, 1,45) — данный термин является технически неправильным, но именно так я называю эту точку (запятую).
- ✓ Табл. 24.2 в главе 24 “Глава о функции `printf()`” содержит список заполнителей для функции `printf()`. Там вы увидите, что `%f` используется для отображения числа с плавающей запятой, типа того, которое появляется в `ICKYGU.C`.
- ✓ Заключительная инструкция `printf()` отображает значение переменной `price` (цена) с плавающей точкой:

```
// ("Это будет $%f, пожалуйста \n ", цена)  
printf("That be $%.2f, please.\n", price);
```

Чтобы сделать это, используется заполнитель `%f` (`f` означает `float` — с плавающей точкой). Однако чтобы с помощью `%f` отобразить значение как сумму денег, требуется некоторое дополнительное форматирование. Чтобы все выглядело профессионально, вы вставляете “точку 2” между `%` и строчной `f`. Это форматирует вывод, оставляя только две десятичных позиции. Поэтому вместо `%f` в строке форматирования используется `%.2f`.

Возможно, вы хотите еще две пинты?

Вы можете легко раскрутить `ICKYGU.C`, чтобы она выполнила некоторые математические действия. Предположим, что вы хотите выяснить, сколько стоят две пинты этого неприглядного оранжевого варева. Сначала вы изменяете значение переменной `pints` (пинты) в строке `int pints=1`; так:

```
int pints=2;
```

Это присваивает переменной `pints` значение 2. Затем нужно указать необходимые математические действия в функции заключительной `printf()`, чтобы вычислить, сколько стоят две пинты этого липкого варева. Сделайте следующие изменения:

```
// ("Это будет $%f, пожалуйста \n ", цена)  
printf("That be $%.2f, please.\n", pints*price);
```

Фактически нужно сделать лишь одно изменение, притом в конце строки. Прежде там была только переменная `price` (цена). Теперь там выражение `pints*price`, т.е. произведение значения цены `price` на значение `pints`. Поскольку цена выражена десятичным числом с плавающей точкой, то и результат будет числом с плавающей точкой. Поэтому в строке форматирования используется заполнитель `%f`.

Сохраните эти изменения и перетранслируйте программу. Вам придется заплатить больше, но ваш желудок отблагодарит вас.

Множественные объявления

В С полно всяких сокращений. Это одна из причин, по которой никакие две программы на С не похожи друг на друга: программисты всегда в своих интересах используют различные способы делать одни и те же вещи. Одна из таких уловок — объявлять несколько переменных в одной инструкции. Я знаю — это обычно рассматривается как нарушение закона в некоторых южных штатах, но это теперь стало законным повсюду в Объединении.

Следующие три инструкции `int` создают три целочисленных переменных: `methus`, `you` и `diff`:

```
int methus;  
int you;  
int diff;
```

Представленная ниже однострочная инструкция делает то же самое:

```
int methus, you, diff;
```

Каждая из переменных определена после ключевого слова и пробела `int`. После каждой переменной стоит запятая, а после последней переменной стоит точка с запятой, заканчивающая инструкцию.

Это сокращение прежде всего позволяет сократить листинг. Оно позволяет объявить все переменные одного типа в одной строке экрана, а не располагать крошечные инструкции `int` в колонку в начале программы.



Во множественном объявлении можно объявить только переменные одного (того же самого) типа. Например:

```
int top, bottom, right, left;  
float national_debt, pi;
```

Целые переменные объявлены на одной строке, а переменные с плавающей точкой (нецелые числа) — на другой.



Если в объявлении переменной присваивается значение, записывайте такое объявление в отдельной строке. Вот так:

```
int first=1;  
int the_rest;
```

Константы и переменные

В языке С предусмотрены не только переменные, но и константы. Они используются подобно переменным, хотя их значения никогда не изменяются.

Предположим, что однажды кто-то обманул вас, уговорив написать программу с тригонометрическими формулами. В программах этого типа часто используется страшное значение π (пи). Оно равно 3,1415926535897932384626433832795028841971693993751 (и еще бесконечно много десятичных знаков). Поскольку оно никогда не изменяется, вы можете создать константу, назвав ее `pi` (пи), и положить ее равной значению π с подходящей точностью.

Другой пример константы — строка текста в кавычках:

```
printf("%s", "This is a string of text"); // "Это - строка текста"
```

Текст `"This is a string of text"` ("Это — строка текста") является константой, используемой данным вызовом функции `printf()`. Конечно, на ее месте могла бы быть и переменная, но в данном примере вместо переменной используется строковая константа — литерал, а именно строка в кавычках.

- ✓ Константа используется точно так же, как и переменная, хотя ее значение никогда не изменяется.
- ✓ Числовая константа — это число, такое как π , константа в программе не изменяется.
- ✓ Символ π читается “пи”. Это греческая буква p . Мы читаем английскую букву p как “пи”.
- ✓ Строковая константа — это текст, который никогда не изменяется, хотя это справедливо для большей части текста в программе на языке C. Однако в этой главе основное внимание уделяется прежде всего числовым константам.

Придумывание и определение констант

Вся эта постоянная ерунда может показаться глупой. Но однажды вы столкнетесь с программой, подобной SPEED.C, — за исключением того, что программа будет намного длиннее, — и вот тогда вы на самом деле оцените значение констант в языке C и полезность директивы `#define`, о которой вы прочитаете далее в этой главе:

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

```
int main() // главная функция
{
    // ("Но ведь ограничение скорости равно - %i.\n ", 55)
    printf("Now, the speed limit here is %i.\n", 55);
    // ("Но вы ехали со скоростью' %i.\n ", 55+15)
    printf("But I clocked you doin' %i.\n", 55+15);
    // ("разве вы не видели этот знак %i МИЛЬ В ЧАС? \n ", 55)
    printf("Didn't you see that %i MPH sign?\n", 55);
    return(0);
}
```

Начните ввод нового файла в редакторе. Тщательно напечатайте предыдущий исходный текст. В нем нет ничего нового или отталкивающе отвратительного. Сохраните файл на диске под именем SPEED.C.

Скомпилируйте его! Исправьте его, если нужно! Выполните его!

Вывод довольно прост; он выглядит так:

```
Now, the speed limit here is 55.
But I clocked you doin' 70.
Didn't you see that 55 MPH sign?
```

(Но ведь ограничение скорости равно 55.

Но вы ехали со скоростью 70.

Разве вы не видели этот знак 55 МИЛЬ В ЧАС?)

Ну и что? Большое дело!

Ну а если бы ограничение скорости было равно 45? Тогда пришлось бы редактировать программу и везде заменить 55 на 45. А если бы программа была длиной 800 строк, как бы вы сделали это? А что, если бы вам пришлось изменить еще несколько других вхождений констант в программу, как бы вы знали, какие константы использовались, и как с помощью вашего редактора вы отыскивали бы каждую константу и заменяли ее должным образом, как бы вы различали, например, месяцы или годы? К счастью, в языке C предусмотрены средства решения этой проблемы.



- ✓ Вы можете с апломбом утверждать, что это — не проблема. В конце концов, большинство редакторов имеет команду поиска и замены. К сожалению, поиск и замена чисел в компьютерной программе — опасная вещь, ее не так

просто сделать! Предположим, что нам нужно найти и заменить число 1. Поиск и замена этого числа изменили бы также и другие значения: 100, 512, 3,141 — все они попались бы в сети поиска и замены.

- ✓ Программа изменится, если ограничение скорости понизить до 45. В конце концов, нарушитель ехал со скоростью 70 или 60 миль в час? С моей точки зрения, это не имеет значения. Если хотите, вы можете исправить программу самостоятельно.

Удобное сокращение

Есть два удобных сокращения для записи числовых констант в C. Я предлагаю два решения. Первое решение состоит в том, чтобы постоянное значение хранить в переменной:

```
int speed=55;
```

Эта строка работает, потому что компилятор сохраняет значение 55 в целочисленной переменной `speed` (скорость), и вы можете тогда использовать `speed` (скорость), а не 55 в вашей программе. Чтобы изменить скорость (значение `speed`), при редактировании нужно сделать только одно изменение:

```
int speed=45;
```

Хотя эта строка решает проблему, это решение совершенно бестолковое, потому что переменная предназначена для хранения значений, которые изменяются. Компилятор должен выполнить всю работу, вроде взбить подушки и создать все условия, необходимые для переменной, а затем вы используете эту переменную всего лишь в качестве константы. Нет, истинное решение состоит в том, чтобы определить постоянное значение как символическую константу. Это в действительности проще, что доказывает модифицированная программа `SPEED.C`:

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

```
#define SPEED 55
```

```
int main()                                // главная функция
{
    // (Но ведь ограничение скорости равно %i.\n ", СКОРОСТЬ)
    printf("Now, the speed limit here is %i.\n",SPEED);
    // ("Но вы ехали со скоростью' %i.\n ", СКОРОСТЬ +15)
    printf("But I clocked you doin' %i.\n",SPEED+15);
    // ("разве вы не видели этот знак %i МИЛЬ В ЧАС? \n ", СКОРОСТЬ)
    printf("Didn't you see that %i MPH sign?\n",SPEED);
    return(0);
}
```

Здесь сделано несколько изменений:

- ✓ Добавлена новая, третья строка программы — одна из тех магических строк, которые начинаются со знака фунта (`#`). Эта строка, `#define`, определяет числовую константу, которая может использоваться в программе повсюду:

```
#define SPEED 55
```

Как и в случае с `#include`, эта строка не заканчивается точкой с запятой. Фактически, строка с `#define` отличается от обычной строки двумя большими ляпами: в этой строке не используется знак `"="` и строка не заканчивается точкой с запятой. Компилятор, конечно же, сгенерирует сообщения об ошибках, если вы забудете хотя бы об одном из этих отличий.

- ✓ Слово `SPEED` используется в программе в трех инструкциях `printf()` для того, чтобы представить значение 55. В инструкции `printf` это слово используется точно так же, как число или переменная.

По секрету расскажу вам, что происходит на самом деле. Компилятор видит имя, указанное в `#define`, и совершенно самостоятельно делает поиск и замену. Когда компилируется вся программа, в инструкции `printf()` будет использовано значение 55. Преимущество состоит в том, что вы можете легко модифицировать постоянные значения, просто редактируя директиву `#define`.

Тщательно отредактируйте исходный текст `SPEED.C`, чтобы он совпал с приведенным выше. Сохраните файл на диске, а затем перетранслируйте. Вывод программы будет тот же самый, потому что единственное изменение свелось к тому, чтобы превратить значение 55 в настоящую константу, а не использовать само значение в программе.

- ✓ Главное преимущество снова заключается в том, что требуется отредактировать только одну строку (и это можно сделать быстро), чтобы изменить ограничение скорости (`speed`). Чтобы сделать это, достаточно отредактировать строку `#define SPEED 55` вот так:

```
#define SPEED 45
```

Фактически вы изменили постоянное значение 45 в трех других местах в программе. Благодаря этому изменению экономится и некоторое время при выполнении программы `SPEED.C`, но главное — экономится гораздо больше времени, затрачиваемого программистом на разработку длинных, гораздо более сложных программ, в которых также используются постоянные значения.

- ✓ Символическая константа в языке C — обозначение постоянного значения, созданного с помощью директивы `#define`.



Директива `#define`

Конструкция `#define` (это ее официальное название, хотя я предпочитаю называть ее директивой) используется для того, чтобы определить в программе то, что профессиональные знатоки C называют символической константой, — сокращенное название (имя) значения, которое появляется много раз в исходном тексте. Вот формат этой конструкции:

```
// #define СОКРАЩЕНИЕ значение
#define SHORTCUT value
```

`SHORTCUT` (СОКРАЩЕНИЕ) — название (имя) определяемой константы. По традиции такие константы называют так же, как и переменные, но ВСЕ БУКВЫ в имени делают ЗАГЛАВНЫМИ (пробелы в имени, конечно, не допускаются).

`value` (значение) — это значение, заменяющее слово `SHORTCUT`. Это слово заменяется своим значением глобально повсюду в остальной части файла исходного текста. Значение может быть числом, выражением, символом и даже строкой.

В конце строки точка с запятой не нужна, но строка непременно должна начинаться со знака фунта. Эта строка должна быть написана в начале исходного текста, перед главной функцией `main()`.

Вы должны также вставить комментарий в конце строки с директивой `#define`, чтобы напомнить о том, что представляет собой значение, — так, как это сделано в следующем примере:

```
#define SPEED 55 /* ограничение скорости */
```

Здесь для SPEED задано значение 55. Теперь, чтобы представить 55, в программе везде можно использовать SPEED.

Строковая константа может быть создана тем же самым способом, хотя это и не столь популярно:

```
// ПОДРУГА "Бренда"  
#define GIRLFRIEND "Brenda" /* моя куклка на этой неделе */
```

Единственное различие состоит в том, что строка заключена в двойные кавычки, что по традиции считается обязательным для строки в C.



- ✓ ВСЕ БУКВЫ в слове-сокращении обычно ЗАГЛАВНЫЕ, чтобы не спутать его с именем переменной. За исключением этого, все остальные правила именования переменных обычно применяются и к словам-сокращениям. Моя рекомендация: давайте им короткие и простые имена.

- ✓ Необходимо следить за типами определяемых констант и затем использовать их в соответствии с их типами, как показано в следующем примере:

```
// ("ограничение скорости равно %i.\n", СКОРОСТЬ)  
printf("The speed limit is %i.\n", SPEED);
```

- ✓ А также в следующем:

```
puts(GIRLFRIEND); // ПОДРУГА
```

- ✓ Эти строки выглядят несколько странно, но они вполне законны, потому что компилятор знает, сокращениями чего являются SPEED и GIRLFRIEND.

- ✓ Знак "=" после слова-сокращения недопустим!

- ✓ Строка, содержащая конструкцию #define, не заканчивается точкой с запятой (это ведь не инструкция языка C)!

- ✓ В определяемой строковой константе можно использовать также escape-последовательность, т.е. символы, перед которыми стоит наклонная черта влево.

- ✓ Строковые константы очень редко определяются с помощью #define. Только если в программе строка используется очень много раз, ее необходимо сделать константой. В противном случае для отображения текста на экране в большинстве программ используются обычные printf() и puts.

- ✓ При определении числовой константы вы можете также использовать математические выражения и другие не менее странные вещи. В данной книге я не углубляюсь в эту тему, но вполне законны строки, подобные следующей:

```
// РАЗМЕР
```

```
#define SIZE 40*35
```

Здесь значение слова-сокращения SIZE устанавливается равным выражению 40*35, а программист даже не вычисляет, чему оно равно.

- ✓ Использовать директиву #define не обязательно, и вас не оштрафуют, если вы не используете ее. Несомненно, вы можете вставлять числа непосредственно в программу. И, конечно же, вы можете использовать переменные, чтобы хранить в них константы. Я не буду сердиться за это. И вас не посадят в тюрьму.



Настоящие постоянные переменные в действии

С помощью #define удобно создавать константы, которые потом можно использовать в программе повсюду. Я рекомендую использовать эту директиву всякий раз, когда в программе есть нечто такое, что может измениться. Например:

```
#define NUMBER_OF_USERS 3
#define COLUMN_WIDTH 80
#define US_STATES 50
```

Каждое из этих объявлений позволяет изменить константы программы, редактируя единственное объявление `#define`. Но действительно ли это константы — противоположность переменных?

Нет! Дело в том, что в языке C предусмотрено ключевое слово `const` (константа), которое преобразовывает короткую переменную в упрямую константу. Вот так:

```
const int senses = 6;           // константа
```

Предыдущая инструкция создает переменную, названную `senses` (чувства), причем устанавливает значение этой переменной равным 6. Данное значение не может быть изменено или использоваться для чего-нибудь другого позже в программе; если вы попытаете это сделать, компилятор сообщит, что это ошибка — данная “переменная” может использоваться только для чтения (“read-only variable”).

Но все же числовые константы (постоянные значения) с помощью ключевого слова `const` создаются не так уж и часто. Однако весьма часто используется инструкция с ключевым словом `const` для создания строковой константы. Вот пример такой инструкции:

```
const char prompt[] = "Your command:";
```

Эта инструкция создает строковую переменную `prompt` (подсказка) и устанавливает ее значение равным `"Your command: "` (“Ваша команда:”). Ключевое слово `const` гарантирует, что содержимое этой переменной не может быть изменено — это особый тип переменной — константа.

Числа в языке C

В этой главе...

- Хранение чисел различных типов в переменных различных типов
- Что такое длинные (long) и короткие (short) числа типа int
- Переменные со знаком и без знака
- Вещественные числа
- Вещественные числа удвоенной точности
- Форматирование больших значений

Новички! Пристегните ваши ремни безопасности! Я слишком долго танцевал в аду пылающих чисел. Пришло время и вам с головой погрузиться в адский огонь значений и цифр. Глубоко внизу под безопасными и удобными в использовании числами типа int залегают большие и отвратительные монстры. Ужасающие значения необходимо очень осторожно размещать в маленьких программах. Числа, даже смертельные и ядовитые, все равно можно приручить, если следовать моим нежным советам из этой главы.

Все причины опасений напрасны! Просто наденьте ваш асбестовый костюм и следуйте за мной. Наблюдайте, как хожу я.

Числа, числа и числа, которыми несть числа

Добро пожаловать в мир чисел! Этот мир вскоре может приоткрыть один из многих новых, печальных аспектов языка программирования C. Он называется Проблемой Типов Числовых данных в C. В отличие от реальной жизни, где мы можем просто взять любое число “с потолка” и, полные счастья, с радостью любоваться им, в C вы можете вытягивать числа только из определенных переменных, притом с учетом типа данного числа. Уже одно это может расстроить, логично будет задать вопрос: “Что такое тип числа?” На самом деле это не “тип числа”, это тип числовых данных; просто те, кто работает в Пентагоне, полюбили термин “тип числа”. Вы должны сообщить компилятору языка C тип числа, потому что он думает о числах совсем не то, что люди. Например, о числе вы должны знать следующее:

- ✓ Действительно ли это целое число, т.е. число без дроби или десятичной части?
- ✓ Насколько оно большое (больше 32 000 или настолько большое, что не влезает на странице)?
- ✓ Если число действительно имеет дробную часть, какова должна быть точность? (Сколько десятичных знаков нужно хранить, можно ли отбросить тысячные, миллионные, или же нужно хранить не менее миллиона десятичных знаков. Ученые должны знать, с какой точностью выполнялись расчеты, чтобы посылаять ракеты странам с противоположной идеологией.)

Я знаю, что этот материал чужд вам. Большинство программистов просто хочет иметь переменные, в которых можно хранить числа до тех пор, пока они не выскочат из задней панели компьютера и не станут правительственной статистической! Но вам придется немного поразмыслить над числами, прежде чем вы научитесь делать это.

- ✓ Самый распространенный числовой тип данных — целое число.
- ✓ Если вы собираетесь работать с десятичными числами, вроде сумм в долларах, вам придется использовать числа с плавающей запятой.
- ✓ Продолжайте читать.

Числа в C

В C предусмотрено множество различных типов чисел. Они называются числовыми типами данных и перечислены в табл. 9.1 вместе с дополнительными сведениями. Приклейте таблицу где-нибудь на видном месте. Время от времени обращайтесь к этой таблице, потому что только абсолютный безумец смог бы запомнить ее.

Таблица 9.1. Числовые типы данных в языке C

Ключевое слово	Тип переменной	Диапазон
char	символ (или строка)	от -128 до 127
int	целое число	от -32 768 до 32 767
short	короткое целое число	от -32 768 до 32 767
short int	короткое целое число	от -32 768 до 32 767
long	длинное целое число	от -2 147 483 648 до 2 147 483 647
unsigned char	символ без знака	от 0 до 255
unsigned int	целое число без знака	от 0 до 65 535
unsigned short	короткое целое число без знака	от 0 до 65 535
unsigned long	длинное целое число без знака	от 0 до 4 294 967 295
float	с плавающей точкой с одинарной точностью (точность 7 цифр)	от $-3,4 \times 10^{38}$ до $-3,4 \times 10^{-38}$ и от $3,4 \times 10^{-38}$ до $3,4 \times 10^{38}$
double	с плавающей точкой с двойной точностью (точность 15 цифр)	от $-1,7 \times 10^{308}$ до $-1,7 \times 10^{-308}$ и от $1,7 \times 10^{-308}$ до $1,7 \times 10^{308}$

- ✓ *Ключевое слово* в таблице — ключевое слово языка C, обычно используемое в объявлении типа переменной. Если вы читали предыдущие главы этой книги, вы уже использовали типы int, char и float.
- ✓ *Тип переменной* в таблице указывает, какой тип переменной определяет ключевое слово. Например, char определяет символьную (или строковую) переменную, а int определяет целые числа. В языке C предусмотрено много типов переменных, каждый из которых зависит от типа числа или описываемого значения.
- ✓ *Диапазон* указывает, какие (по величине) числа может хранить переменная данного типа. Например, целые числа должны лежать в диапазоне от -32 768 до 32 767. Для обработки больших значений используются другие типы переменных. В принципе, диапазон значений, предусмотренный в вашем компиляторе, может отличаться; поэтому диапазоны значений, приведенные в табл. 9.1, используйте только для справки.

Таблица 9.1 не так уж и сложна. В С фактически предусмотрено только четыре типа переменных:

- ✓ char (символ);
- ✓ int (целое число);
- ✓ float (с плавающей точкой);
- ✓ double (двойной точности).

Тип int может модифицироваться с помощью слова short (короткое) или long (длинное), и, кроме того, и char, и int может модифицироваться с помощью unsigned (без знака). А значения переменных типа float и double относятся к числам с плавающей точкой; отличие между этими типами состоит в том, что тип double имеет больший диапазон значений.

Зачем использовать целые числа? Почему нельзя ограничиться только числами с плавающей запятой?

Казалось бы, если есть возможность обрабатывать числа с плавающей запятой с двойной точностью (тип double) с таким большим диапазоном значений, зачем тогда объявлять, что в некоторых переменных будут храниться целые числа? Казалось бы, достаточно объявить все переменные как double (двойной точности с плавающей запятой) и можно забыть обо всех остальных типах! Это хорошо звучит, на самом деле все гораздо хуже.

Целые числа — действительно самые удобные и наиболее часто употребляемые типы числовых переменных. Часто в программе нужны только маленькие целые значения. Числа с плавающей запятой хороши, но компьютеру их обрабатывать сложнее и дольше. По сравнению с ними обработка целых чисел выполняется намного быстрее. По этой причине Бог счел целесообразным создать целые числа. (Между прочим, Он сделал это на третий день.)

Целочисленные типы (короткий — short, длинный — long, широкий — wide, жирный — fat и т.д.)

Вас должны интересовать только два типа целых чисел: обычное целое число — int — и длинное целое число — long. (Числа со знаком и без знака (unsigned) будут обсуждаться далее в этой главе.) Тип int определяет значения целого числа, обычно в пределах от -32 768 до 32 767. Он идеально подходит для маленьких чисел без дробной части. В некоторых версиях С этот тип значений обозначается как short (короткое) или short int (короткий int).

Тип long (длинный) задает значения целого числа в пределах от -2 147 483 648 до 2 147 483 647 — большой диапазон, но не достаточно большой для национального долга или эго Мадонны. В некоторых версиях С этот тип числовой переменной обозначается как long или long int.

- ✓ Ключевые слова int и long позволяют объявить целые переменные. int используется для меньших значений, а long — для больших значений.
- ✓ Для отображения значений переменных типа int в функции printf() используется метка-заполнитель %d. (Можно также использовать метку-заполнитель %i; подробности вы найдете в табл. 24.2 в главе 24 “Глава о функции printf()”.)
- ✓ int = short = short int.
- ✓ long = long int.



- ✓ Целые переменные (int) короче, обрабатываются компьютером быстрее и проще. Используйте тип int для маленьких целочисленных значений.
- ✓ В некоторых компиляторах языка C диапазоны для int и long int совпадают. Дело в том, что компилятор (обычно 32-разрядная модель) может более эффективно обработать значения типа long int, чем меньшие значения int. Это просто технические детали; не старайтесь их запомнить и тем самым испортить себе день.
- ✓ Отрицательные числа — о чем беспокоиться? Иногда они нужны, но в большинстве случаев можно о них не думать. (См. следующий раздел.)
- ✓ Тип переменной char может использоваться также как тип целого числа, хотя он имеет чрезвычайно маленький диапазон. Переменные такого типа главным образом используются для того, чтобы сохранить отдельные символы (или строки); это мы еще обсудим в главе 10 “Переменные типа char”.

Со знаком или без знака (unsigned), или может ли перед этим стоять знак “минус”?

Я очень часто обхожусь без отрицательных чисел. Они нужны только тогда, когда вы имеете дело с алгеброй. Но есть программы, в которых без отрицательных чисел не обойтись.

Когда в C вы объявляете числовую переменную, у вас есть выбор: *со знаком* или *без знака* (unsigned). Все будет *со знаком*, если перед типом переменной вы не напечатали unsigned (*без знака*):

```
unsigned int shoot_the_moon = 26;
```

Значение переменной численного типа со знаком может быть отрицательным. Значение переменной стандартного типа int может находиться в диапазоне от -32 768 до 32 767. Этот диапазон включает отрицательные числа (от -32 768 до -1), положительные числа (от 1 до 32 767), а также нуль. (Нуль и положительные числа образуют множество неотрицательных чисел.)

Значения переменной численного типа без знака (unsigned) могут быть только положительными или нулем. Для чисел без знака диапазон смещается в сторону положительных чисел, так как отрицательные значения не допускаются. Обычный тип int без знака (unsigned int) имеет диапазон от 0 до 65 535. Отрицательные числа не допускаются.

Пусть переменная elephants (слоны) типа int имеет значение 40 000. Попробуйте втиснуть это значение в тип int со знаком! Ха!

```
unsigned int elephants=40000;
```

В табл. 9.2 иллюстрируются различия между типами переменных; в ней указаны значения, которые могут храниться в различных типах.

Таблица 9.2. Диапазоны значений переменных со знаком и без знака

Со знаком	Диапазон	Без знака	Диапазон
char	от -128 до 127	unsigned char	от 0 до 255
int	от -32 768 до 32 767	unsigned int	от 0 до 65 535
long	от -2 147 483 648 до 2 147 483 647	unsigned long	от 0 до 4 294 967 295 до 2 147 483 647



- ✓ Числа с плавающей запятой (числа с десятичной точкой или дробями) могут быть положительными или отрицательными независимо от всей этой ерунды, касающейся знаков.
- ✓ Числа с плавающей запятой рассматриваются в следующем разделе.
- ✓ Обычно различия между значениями со знаком и без знака (unsigned) не существенны.
- ✓ Переменные со знаком могут раздражать и служить источником всевозможных расстройств, порождая жуткие ошибки. Вот как это все может происходить: предположим, что вы добавляете 1 к целочисленной переменной со знаком. Если эта переменная уже содержит значение 32 767, ее новое значение (после того, как вы добавите 1) будет равно -32 768. Да, даже при том, что вы добавляете положительное число, результат может оказаться отрицательным. В данном случае, чтобы избежать проблемы, лучше было бы использовать переменную типа unsigned int.
- ✓ Чтобы использовать переменную без знака (unsigned), в объявлении этой переменной нужно указать ключевое слово unsigned int или unsigned long. Ваш компилятор языка C может иметь секретный переключатель, который позволяет создавать программы с переменными без знака; обратитесь к интерактивной справочной системе за подробностями.



Как в компьютере представлены целые числа со знаком

Рассматривается ли целое число со знаком или без знака (unsigned) зависит от того, как число хранится в компьютере. Секрет в том, что все данные, независимо от того, как они выглядят на экране или в программе, в компьютере хранятся в двоичном коде (языке). Это система счисления с основанием 2. В ней всего две цифры — 1 (единица) и 0 (нуль).

Двоичные числа составлены из битов, или двоичных цифр. Предположим, что ваш компилятор языка C использует два байта для хранения целого числа. Эти два байта содержат 16 двоичных цифр, или битов. (В байте восемь битов.) Например:

0111 0010 1100 0100

Это значение в десятичной системе (система, которой пользуемся мы в повседневных расчетах) записывается как число 29 380. В двоичной системе единицы и нули представляют различные степени двойки, что может быть весьма сложным для вас, но для компьютера пересчитать их — все равно, что для вас съесть мороженое.

Рассмотрим следующее число:

0111 1111 1111 1111

Это значение 32 767 почти полностью заполнено единицами. Если вы добавите 1 к этому значению, вы получите следующее удивительное число:

1000 0000 0000 0000

То, как компьютер интерпретирует это двоичное значение, зависит от определения переменной. Если он рассматривает ее как значение со знаком, единица в крайней левой позиции числа вообще не рассматривается как цифра 1. Это знак "минус". Само число в этом случае считается равным -32 768 в двоичной системе. Если переменная содержит значение без знака (unsigned), это представление интерпретируется как положительное число 32 768.

Таким образом, интерпретация числа (без знака (unsigned) или со знаком) зависит от того, как рассматривается этот ужасный первый бит в двоичном языке компьютера. Если число со знаком, первый бит рассматривается как знак “минус”. В противном случае первый бит — это только еще один бит в компьютере, притом самый старший, а вовсе не знак “минус”.

Как объявить число с плавающей запятой

Два совка мороженого....

Целочисленные переменные — рабочие лошади в ваших программах, именно они позволяют решить большинство численных задач. Однако когда нужно выполнять действия над дробями, числами, которые имеют десятичные знаки после запятой, или очень большими значениями, нужны числовые переменные другого типа. Это переменные с плавающей точкой.

Ключевое слово `float` (с плавающей точкой) позволяет отвести место для переменной с плавающей точкой, которая может содержать нецелые числа. Вот формат такого объявления:

```
// переменная величина с плавающей точкой;  
float var;
```

После ключевого слова `float` идет пробел или табуляция, а затем — имя переменной, например `var`. В конце строки ставится точка с запятой.

Вы можете также объявить переменную `var` с плавающей точкой (`float`) и присвоить ей значение `value` таким же способом, как и любой другой переменной в C:

```
float var=value;
```

В этом формате после переменной `var` идет знак “=”, а за ним — значение `value`, которое будет присвоено переменной.

Float (с плавающей точкой) — просто сокращение для *floating point* (плавающая точка/запятая). Этот термин относится к десятичной точке в числе. Например, следующее число — значение с плавающей точкой:

```
123.4567
```

Целочисленная переменная не подошла бы для хранения этого числа. В ней можно было бы хранить только 123 или 124. Но десятичное число можно хранить только в переменной с плавающей точкой.

Диапазон для чисел с плавающей запятой является весьма большим. Большинство компиляторов языка C позволяет хранить любое число в диапазоне от $-3,4 \times 10^{38}$ до $3,4 \times 10^{38}$ и от $3,4 \times 10^{-38}$ до $3,4 \times 10^{-38}$. Это огромный диапазон значений (10^{38} — это 1 с 38 нулями). Конечно, большинство чисел, используемых в реальных программах, гораздо меньше.

- ✓ Правила именования переменных перечислены в главе 8 “Переменные в языке C”.
- ✓ Нецелые значения (числа) хранятся в переменных с плавающей точкой.
- ✓ Хотя 123 — целочисленное значение, его все же можно хранить в переменной с плавающей точкой. Однако же...
- ✓ Переменные с плавающей точкой (`float`) должны использоваться только в случае необходимости. Они требуют большего объема оперативной памяти и большего времени обработки, чем целые числа. Если можете обойтись целыми числами, используйте переменные целочисленного типа вместо вещественных чисел.

Давайте напишем программу, обрабатывающую числа с плавающей запятой!

Предположим, что вы и я — огромные, выпуклоголовые создания с планеты Редмонд, все слизистые и зеленые. Мы летаем на нашем НЛО по всей галактике, пьем синее пиво и пишем программы на С для наших компьютеров. Я — Дэн. Ваше имя — Карл.

Однажды при нападении на коров в штате Индиана мы начинаем следующий диалог:

Дэн: Световой год равен 5 878 000 000 000 миль! Это 5 триллионов 878 миллиардов плюс какая-то мелочь! Я не иду на это!

Карл: Нет, это всего лишь каких-то 483 400 000 миль от Солнца до Юпитера. Это всего лишь ничтожная доля светового года.

Дэн: Какая доля?

Карл: Хорошо, почему бы тебе не напечатать следующую программу на С и заставить компьютер вычислить это расстояние?

Дэн: Подожди, я — автор этой книги. Это ты должен напечатать программу JUPITER.C и выяснить все детали.

Вот программа.

```
#include <stdio.h>

int main()                                // главная функция
{
    float lightyear=5.878E12;
    float jupiter=483400000;
    float distance;                        // расстояние
    distance=jupiter/lightyear;
    // ("Юпитер на расстоянии %f световых лет от солнца \n", расстояние)
    printf("Jupiter is %f light years from the sun.\n",distance);

    return(0);
}
```

Введите эту программу с помощью вашего текстового редактора. Будьте внимательны! Проверьте орфографию, всякие странные символы и все остальное. Сохраните файл на диске под именем JUPITER.C.

Скомпилируйте программу. Если будут какие-либо ошибки, исправьте их и перетранслируйте программу.

Выполните программу. Результат будет таким:

Jupiter is 0.000082 light years from the sun.

(Юпитер на расстоянии 0,000082 светового года от Солнца.)

Карл: Просто потрясающе!

Дэн: Я все равно против.

- ✓ Переменная с плавающей точкой объявляется с помощью ключевого слова `float`.
- ✓ В экспоненциальном формате, который используется учеными для упрощения печатания нулей и запятых, длина светового года записана как `5.878E12`. Это означает, что в десятичном числе `5,878` запятая должна быть сдвинута вправо на 12 позиций. (Этот уродливый экспоненциальный формат (с буквой `E` внутри числа) описан в следующем разделе.)
- ✓ Значение переменной `jupiter` (Юпитер) установлено равным среднему расстоянию между Юпитером и Солнцем, которое равно 484 миллионам миль. В исходном тексте после `4834` следует 5 нулей. Здесь не нужно вносить беспорядок экспоненци-

альным форматом, потому что компилятор может прочитать это относительно небольшое число. (А вот числа большие 100 миллиардов обычно нужно записывать в экспоненциальном формате (с буквой E внутри числа); однако точная величина границы должна быть указана в руководстве к вашему компилятору.)

- ✓ Переменная `distance` (расстояние) содержит результат деления расстояния между Солнцем и Юпитером на длину светового года. Иными словами, она равна количеству световых лет от Юпитера до Солнца. Полученная дробь является чрезвычайно маленькой.
- ✓ Чтобы отобразить значение с плавающей точкой, в функции `printf()` используется метка-заполнитель `%f`.
- ✓ Переменные с плавающей точкой (`float`) используются в этой программе по двум причинам: из-за величины чисел и потому что в результате деления обычно получается не целое число, а десятичная дробь.

Экспоненциальное представление (с буквой E)

Когда приходится иметь дело с очень большими и очень маленькими числами, используется старое экспоненциальное представление (с буквой E). Я предполагаю, что нужно обсудить эту тему, потому что если вы интересуетесь программами, в которых используются эти типы чисел, вы, вероятно, уже слышали о них.

Экспоненциальное представление (с буквой E) требуется в C (или даже в электронной таблице Excel), когда некоторые числа становятся невероятно огромными. Такие числа — это числа с плавающей запятой. Целые числа в программах такими большими не бывают.

Если число имеет около восьми или девяти цифр, оно должно быть выражено в экспоненциальном представлении (с буквой E), в противном случае компилятор не “проглотит” его. Например, рассмотрим длину светового года в милях:

```
5 878 000 000 000
```

Это 5 триллионов 878 миллиардов. В языке C в записи числа запятые, точки и пробелы не допускаются, поэтому число нужно записать следующим образом:

```
5878000000000
```

Это цифры 5878, за которыми идет девять нулей. Значение, конечно, то же самое; однако для удобства чтения добавляются пробелы или запятые, чтобы разбить большие числа на классы (в программе их приходится удалять). И хотя это число попадает в диапазон чисел с плавающей точкой, скомпилировать его компилятор не может, потому что оно является слишком длинным (содержит слишком много цифр). Причина ошибки компилятора — не значение, а количество цифр в числе.

Чтобы компилятор мог вычислить значение, его нужно выразить так, чтобы в записи было меньше цифр, и тут на помощь приходит экспоненциальный формат. Вот то же самое значение в экспоненциальном представлении (с буквой E) в том виде, в котором оно определено в программе JUPITER.C из предыдущего раздела:

```
5.878E12
```

В научном, или экспоненциальном представлении (с буквой E) число записывается в следующем формате:

```
x. xxxxEll
```

`x.xxxx` — значение; это одна цифра, после которой идет десятичная точка, после которой, в свою очередь, следует еще несколько цифр. Затем пишется прописная (большая) буква E, а после нее — еще одно значение (`ll`). Чтобы узнать истинную величину числа, десятичную точку в значении `x.xxxx` нужно сдвинуть вправо на `ll` позиций. На рис. 9.1 показано, как найти значение светового года.

5.878	E12
58.78	E11
587.8	E10
5878.	E9
58780.	E8
587800.	E7
5878000.	E6
58780000.	E5
587800000.	E4
5878000000.	E3
58780000000.	E2
587800000000.	E1
5878000000000.	E0
5 878 000 000 000	

Рис. 9.1. Экспоненциальный формат и световой год

Когда в компилятор вы вводите числа в экспоненциальном представлении (с буквой E), используйте надлежащий экспоненциальный формат (с буквой E). Для отображения чисел в экспоненциальном формате (с буквой E) с помощью `printf()` служит метка-заполнитель `%e`. Чтобы увидеть, как она работает, замените `%f` в программе JUPITER.C на `%e`, сохраните измененный файл на диске, перетранслируйте его и выполните полученную машинную программу. В выводе будет использован экспоненциальный формат (с буквой E), получится нечто подобное следующему:

Jupiter is 8.223886e-05 light years from the sun.

(Юпитер на расстоянии 8,223886e-05 световых лет от Солнца.)

Если перед E стоит отрицательное число, как показано в данном примере, десятичную точку нужно перенести влево на *nn* позиций. Вообще, отрицательное значение *nn* указывает на то, что число очень маленькое. Действительно, предыдущее значение равно следующему: .00008223886

- ✓ Экспоненциальный формат (с буквой E) требуется тогда, когда числа содержат так много цифр, что даже компилятор языка C не может их проглотить.
- ✓ Отрицательное число в экспоненциальном формате (с буквой E) после буквы E означает, что значение является очень маленьким. Не забудьте переместить десятичную точку влево, а не вправо, когда вы увидите этот тип числа.
- ✓ В некоторых компиляторах допускается использовать метку-заполнитель `%E` (с прописной (большой) буквой E) в `printf()`, чтобы отобразить числа в экспоненциальном формате с прописной (большой) буквой E.

Диапазон чисел с плавающей точкой удвоенной точности (double) — еще больше, чем у чисел с плавающей точкой одинарной точности (float)!

Для того чтобы обрабатывать действительно огромные числа, в С предусмотрен еще один (на этот раз уже наибольший) тип данных, `double` (с удвоенной точностью). Переменные этого типа могут содержать огромные значения и должны использоваться только тогда, когда вам приходится обрабатывать астрономические значения или когда требуется выполнить математические операции с невероятно высокой точностью.

Переменные двойной точности объявляются с помощью ключевого слова `double` (двойной точности). *Double* (двойной) происходит от термина *double precision* (двойная точность), который означает, что точность таких чисел вдвое выше, чем у обычных чисел с плавающей точкой, которые называются также числами с одинарной точностью.

Что же такое точность? Этот термин относится к десятичным числам, дробям, — и очень маленьким, и огромным числам, — хранящимся в компьютере. Чтобы хранить информацию, компьютер использует только единицы и нули. Для целых чисел этого вполне достаточно. Для нецелых чисел это означает, что имеет место некоторый обман. Этот обман вначале незаметен, но точность результатов уменьшается, они становятся “нечеткими” через некоторое время, особенно в последних цифрах.

В качестве примера рассмотрим число:

123.4567891234

Конечно, это число с плавающей точкой (`float`). И если в С вы присвоите его в качестве значения переменной с плавающей точкой (`float`), компьютер может сохранить его только как значение с одинарной точностью. В этом случае компьютер сможет сохранить только первые восемь цифр. Остальные он отбросит! Точно так же он поступит и с числом

123.45678422231

Первые восемь цифр точны, причем они совпадают у обоих чисел. А остальные? После этих восьми цифр значения различаются. Вот к чему в действительности приводит одинарная точность. Двойная точность может позволить хранить в точности 12 или 16 десятичных знаков (но последующие знаки также неопределенны).

Любая история имеет мораль, и эта история — не исключение. Во-первых, если с помощью компьютера нужно выполнить вычисления над операндами с плавающей точкой, помните, что число может быть представлено только с определенной точностью. Точны будут только приблизительно первые восемь цифр, остальная часть вывода будет бессмысленна. Во-вторых, если вы нуждаетесь в более точных вычислениях, используйте переменные типа `double` (с удвоенной точностью). Конечно, проблемы с точностью могут возникнуть и в этом случае, но значения будут представлены более точно, чем в случае типа `float`.

- ✓ Чтобы в программе на С объявить переменную с плавающей точкой с двойной точностью, используйте ключевое слово `double`.
- ✓ Если вы печатаете значение, например, 123,456, а на выводе видите нечто вроде 123,456001, то дополнительные цифры 001 возникают из-за недостатка точности в компьютерном представлении чисел с плавающей запятой.



В большинстве случаев любые дополнительные цифры являются незначимыми, так что не позволяйте компьютеру обмануть вас.

- ✓ Точность в восемь цифр более чем достаточна для большинства вычислений с нецелыми числами. Однако для того, чтобы послать людей на Марс, я рекомендую вести вычисления с удвоенной точностью (тип `double`). (Я знаю, что НАСА пристально читает мои книги.)
- ✓ В некоторых компиляторах предусмотрены числа учетверенной точности с их собственными уникальными ключевыми словами и другими правилами и инструкциями.
- ✓ Чем больше точность, тем дольше компьютер обрабатывает числа. Не используйте точность большую, чем необходимо.

Форматирование нулей и других знаков десятичного числа

Значения с плавающей точкой могут быть отображены с помощью `%f` в строке функции `printf()`, но при этом они могут выглядеть ужасно большими. Чтобы избежать этой проблемы, придется изучить способы форматирования с помощью `printf()`. К счастью, это достаточно сделать всего лишь один раз.

Между `%` и `f` можно вставить некоторые специальные символы форматирования. Они управляют выводом функции `printf()` и могут помочь избавляться от некоторых дополнительных нулей или урезать отображаемое число.

Следующие несколько примеров покажут, как урезать числа с помощью `printf()` и избежать кавалькады нулей, которая иногда появляется у чисел с плавающей точкой одинарной и двойной точности.



Следующая метка-заполнитель отображает число с плавающей точкой (типа `float`), причем дробная часть занимает только две десятичных позиции. Это идеально подходит для отображения сумм в долларах. Без нее в качестве цены могло бы отобразиться что-нибудь вроде \$199,9500, что выглядело бы крайне странно для ваших клиентов.

`% .2f`

Если после точки необходимо отобразить больше десятичных знаков, укажите их количество:

`% .4f`

Эта метка-заполнитель форматирует числа с плавающей запятой так, чтобы после десятичной точки отобразить четыре цифры. Если значение недостаточно мало, нули заполняют недостающие из четырех десятичных позиций.

Следующий формат указывает, что `printf()` должна отобразить число, используя только шесть позиций для цифр и десятичной точки:

`%6f`

Независимо от того, насколько число большое, оно всегда отображается в шести позициях для цифр (и точки). Но слева вывод дополняется пробелами, а не начальными (ведущими) нулями. Число 123 отображается как

123.

Эта строка начинается с двух пробелов, т.е. напечатана с отступом в два пробела.

Иногда %f может отобразить число примерно так:

145000.000000

В этом случае вы можете урезать число, используя либо формат %.2f, который отображает только два нуля после десятичной точки, либо нечто подобное %6f, что ограничивает вывод только шестью цифрами.

- ✓ Вместо того чтобы указывать числа и другие символы между % и маленьким f, можно использовать метку-заполнитель %e. Она отображает числа в экспоненциальном формате; обычно вывод в этом формате короче, чем тот, который получается с помощью метки-заполнителя %f.
- ✓ Кроме того, есть метка-заполнитель %g. Она отображает число с плавающей запятой или в формате %f или в экспоненциальном (научном) формате %e, в зависимости от того, который из них дает более короткий вывод.

Да, я знаю, что в этой главе совсем немного примеров. Но числа скучны. Так что займемся чем-нибудь более интересным.

Переменные типа char

В этой главе...

- Использование объявления char
- Присваивание символов переменным
- Функции `getchar()` и `putchar()`
- Обработка символа (типа char) как крошечной целой переменной

*М*не кажется, что слишком многие книги по языку C фиксируют внимание на числах и полностью избегают рассмотрения другого типа переменных — переменных, в которых можно хранить символы или строки. Эти переменные определенно очень забавны. Вместо того чтобы содержать значения — значения, а что же еще! — символьные переменные содержат отдельные символы или буквы (символы алфавита) и даже целые текстовые строки. Это, конечно, открывает невиданные возможности для воплощения творческого потенциала отдельных графоманов.

В этой главе завершается ваше плавание по морю объявлений переменных, причем вам будет официально представлен тип переменных `char`. Переменные этого типа могут хранить и отдельные символы, и строки текста. В этой главе рассматривается только хранение отдельных символов.

Еще один тип переменной: char

Хотя я говорю и об отдельном символе, и о строковых переменных, в языке C предусмотрен только один тип переменной: `char`, который определяется с помощью ключевого слова `char`. И я думаю, хотя я не уверен, что это слово лучше читать “кар” или как “символ”, а не как “чар”.

Односимвольные переменные: переменные, в которых хранится один символ

Подобно строке текста, односимвольная переменная объявляется с помощью ключевого слова `char`. В отличие от строки, односимвольная переменная содержит только один символ, но не больше. В некотором смысле символьная переменная походит на расширяемую ячейку. Строковая переменная — это просто несколько ячеек, расположенных друг за другом.

Чтобы отвести память для односимвольной переменной, используется ключевое слово `char`. Вот формат объявления:

```
char var;
```

Ключевое слово `char` пишется обязательно на нижнем регистре, за ним следует пробел, а затем — название (имя) создаваемой переменной (переменная величина в данном случае названа именем `var`).

В следующем формате предусмотрено предварительное определение односимвольного значения:

```
char var = 'c';
```

За ключевым словом `char` следует пробел. За названием (именем) создаваемой переменной (в данном примере переменная величина названа `var`) следует знак "=", а за ним — символ в одинарных кавычках. Инструкция заканчивается точкой с запятой.

В одинарные кавычки заключается отдельный символ, который присваивается переменной. В данном примере присваивание выполнено переменной величине `var`. Вы можете указать любой (единственный) символ или одну из escape-последовательностей, если нужно определить непечатаемый символ, такой как кавычка (двойная или одинарная) или новая строка (символ, который вводится в результате нажатия клавиши <Enter>). (Полный список escape-последовательностей приведен в главе 24 "Глава о функции `printf()`" (табл. 24.1); дополнительные сведения содержатся также во врезке "Ввод некоторых труднодоступных символов".) Односимвольная переменная идеальна для того, чтобы читать один символ (очевидно, только один символ!) с клавиатуры. Используя еще неизвестные чудеса языка C, вы можете сравнить считанный символ с другими символами и заставить программу сделать невиданные вещи. Именно так работает система меню, которая позволяет печатать односимвольные команды, не нажимая <Enter>, и именно с помощью этого средства вы можете написать ваши собственные программы, читающие символы с клавиатуры. Это может быть забавно.



Ввод некоторых труднодоступных символов

Некоторые символы либо не могут быть введены с клавиатуры, либо заданы как escape-последовательности. Например, расширенные ASCII-символы используются в большинстве ПК — они включают символы псевдографики для таблиц, математические символы, а также некоторые иностранные символы. Однако чтобы ввести их в символьные переменные, требуется некоторое дополнительное усилие. Конечно, ввести их вполне возможно, — только требуется немного технической работы. Выполните следующие шаги:

1. Найдите секретное значение кода символа — его ASCII-код или расширенный ASCII-код.
2. Запишите найденное число (код) в шестнадцатеричной системе (системе с основанием 16). (Именно поэтому в таблицах ASCII и диаграммах обычно приводятся шестнадцатеричные значения.)
3. Укажите полученное двуцифровое шестнадцатеричное значение в качестве escape-последовательности (т.е. после `\x`).
4. Не забудьте заключить полученную escape-последовательность — длиной в четыре символа — в одинарные кавычки.

Предположим, что вы хотите использовать в программе символ Британского фунта, £. Секретный номер кода этого символа — 156. Ищите его в приложении Б "Таблица ASCII". Вы увидите, что шестнадцатеричное значение равно 9C. (Шестнадцатеричные числа содержат символы.). Поэтому в программе нужно определить следующую escape-последовательность:

```
'\x9C'
```

Обратите внимание, что она заключена в одинарные кавычки. Символ C или любой другой шестнадцатеричный символ можно напечатать на верхнем или нижнем регистре. Когда escape-последовательность присваивается символьной переменной, компилятор с языка C преобразовывает указанное шестнадцатеричное число в символ — например £, — который и будет храниться в переменной, пока не понадобится.

Символ в действии

Следующая инструкция создает в программе символьную переменную `ch` (можно, конечно, также предопределить значение переменной в случае необходимости):

```
char ch; // символ ch;
```

Эта инструкция создает символьную переменную `x` и присваивает ей символьное значение 'X' (прописная (большая) буква X):

```
char x='X'; // символ x='X';
```



Чтобы присвоить символ односимвольной переменной, используйте одинарные кавычки. Это обязательно!

Некоторые символы вы не можете напечатать на клавиатуре. Например, чтобы определить переменную и присвоить ей в качестве значения символ табуляции (<Tab> на клавиатуре), используйте `escape`-последовательность:

```
char tab='\t'; // символ табуляции <Tab>
```

Эта инструкция создает символьную переменную `tab` (табуляция) и помещает в эту переменную символ табуляции, представленный `escape`-последовательностью `\t`.



- ✓ Односимвольные переменные создаются с помощью ключевого слова `char` (символ).
- ✓ Не используйте квадратных скобок в объявлении односимвольных переменных.
- ✓ Если значение переменной предопределяется символом, заключите этот символ в одинарные кавычки. Не используйте двойные кавычки.
- ✓ Символьной переменной можно присвоить почти любое символьное значение. Специальные, сверхъестественные и другие символы могут быть заданы как `escape`-последовательности.
- ✓ Создание строковых переменных описано в главе 4 "Что такое ввод-вывод?". В данной же главе прежде всего описываются односимвольные переменные.

Запихивая символы в символьные переменные...

Символьной переменной можно присвоить значение одним из нескольких способов. Первый способ состоит в том, чтобы присвоить ей символ так, как вы присваиваете значение числовой переменной. Если `key` (клавиша) — символьная переменная, вы можете присвоить ей в качестве значения символ 'T' с помощью следующей инструкции:

```
key='T'; // клавиша='T';
```

Символ `T`, который должен быть в одинарных кавычках, проскальзывает через знак "=" в бункер, отведенный для односимвольной переменной `key`. Инструкция заканчивается точкой с запятой. (Я предполагаю, что ранее в программе была определена символьная переменная `key` с помощью инструкции `char key;`.) Впрочем, в качестве значения можно указать не только отдельный символ, но и любую `escape`-последовательность (это `\`, за которой идут символы), — в общем, всякую всячину. Важно лишь, чтобы после компиляции это был только один символ.

Есть еще один способ "запихнуть" символ в односимвольную переменную. Он состоит в том, чтобы значение одной символьной переменной присвоить другой. Предположим, что `old` (старый) и `new` (новый) — символьные переменные. Тогда можно написать следующее:

```
old=new;
```

Символ из символьной переменной `new` перелетает через знак "=" и приземляется в символьной переменной `old`. После выполнения предыдущей инструкции обе переменные содержат тот же самый символ. (Это — операция копирования, а не перемещения.)

- ✓ Односимвольным переменным в качестве значения можно присвоить отдельный символ. Но...
- ✓ Знак "=" не помещает строку текста в строковую переменную. А жаль. Но компилятор все равно этого не понимает.

Чтение и запись отдельных символов

В этой книге описаны две функции языка C, обычно применяемые для ввода текста с клавиатуры: `scanf()` и `gets()`. Обе читают текст с клавиатуры; обычно они читают полные предложения. Однако `scanf()` обаятельнее и гибче: она может использоваться и для чтения отдельных символов. Вот как это делается:

```
scanf("%c",&key);
```

В этом формате `scanf()` читает с клавиатуры один отдельный символ, указанный меткой-заполнителем `%c`. Его значение сохраняется в символьной (типа `char`) переменной `key` (клавиша), которая должна быть объявлена в программе ранее. Вот пример программы:

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

```
int main() // главная функция
{
    char key;

    // Напечатайте ваш любимый символ на клавиатуре
    puts("Type your favorite keyboard character:"); scanf("%c",&key);
    // ("Ваш любимый символ - %c!\n ", клавиша)
    printf("Your favorite character is %c!\n",key);
    return(0);
}
```

Внимательно введите этот исходный текст с помощью вашего редактора. Сохраните его на диске под именем `FAVKEY1.C`.

Скомпилируйте и выполните. Вот типичный вывод:

```
Type your favorite keyboard character:
```

(Напечатайте ваш любимый символ на клавиатуре:)

Нажмите клавишу, например **M** (или вашу любимую), а затем нажмите клавишу `<Enter>`.

Вы увидите вот что:

```
M
Your favorite character is M!
```

(Ваш любимый символ — M!)

С клавиатуры читается **M**, прочитанный символ сохраняется в символьной (типа `char`) переменной, а затем отображается функцией `printf()`.

- ✓ Чтобы закончить ввод, вы должны нажать клавишу `<Enter>`, — это особенность работы функции `scanf()`.
- ✓ Вы можете напечатать целую строку текста, но `scanf()` прочтает только первый введенный символ, он и будет считаться вашей любимой клавишей.

Функция getchar()

К счастью, для чтения с клавиатуры отдельной клавиши служит не только `scanf()`. В языке C имеется также функция `getchar()`, которая специально предназначена для чтения отдельных символов с клавиатуры. Вот формат инструкции:

```
var=getchar();
```

Переменная величина `var` — символьная переменная. В ней можно хранить любой символ, введенный с клавиатуры. После имени переменной величины следует знак "=", а после него — `getchar` с двумя круглыми скобками, внутри которых ничего нет. Эта строка — законченная инструкция, поэтому она завершается точкой с запятой.

Функция `getchar()` заставляет программу сделать паузу и ждать нажатия клавиши на клавиатуре. `getchar()` просто сидит и ждет. Сидит и ждет. Сидит. Ждет. Сидит. Ждет. Когда клавиша будет нажата, а затем будет нажата клавиша <Enter>, "значение" символа перескочит через знак "=" и будет сохранено в символьной переменной.

Следующая программа — обновление FAVKEY1.C, на сей раз в ней противная функция `scanf()` заменена лучшей функцией `getchar()`:

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

```
int main()                                // главная функция
{
    char key;

    // Напечатайте ваш любимый символ на клавиатуре
    puts("Type your favorite keyboard character:");
    key=getchar();
    // ("Ваш любимый символ - %c!\n", клавиша)
    printf("Your favorite character is %c!\n",key);
    return(0);
}
```

Отредактируйте исходный текст FAVKEY1.C: замените только функцию `scanf()` функцией `getchar()`, как показано выше. Сохраните новый файл на диске под именем FAVKEY2.C. Скомпилируйте и выполните!

Вывод тот же самый, что и в первой версии программы; и вы все равно должны нажимать клавишу <Enter>, чтобы ввести значение вашей любимой клавиши.

- ✓ Да, кажется глупым, что необходимо нажать еще и <Enter>, для того чтобы ввести единственный символ. Но так работает `getchar()`.
- ✓ Можно читать данные с клавиатуры и в более интерактивном режиме, где нажимать клавишу <Enter> не требуется. Соответствующие методы вы найдете в моей книге *C All-in-One Desk Reference For Dummies*, выпущенной издательством "Wiley".

Функция putchar()

Функция `putchar()` — читается "пут кар", что означает "put character" ("поместить символ"), — является противоположностью функции `getchar()`; `getchar()` читает символ с клавиатуры, а `putchar()` отображает символ на экране. Сейчас я приведу формат инструкции, хотя вы, вероятно, и сами могли бы угадать его:

```
putchar(c);
```

Здесь *c* может быть односимвольной переменной или символьной константой в одинарных кавычках:

```
putchar('c');
```

Символ 'с' будет отображен на экране.

Следующая программа показывает, как с помощью `putchar()` можно “швырять” символы на экран:

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

```
int main()                                // главная функция
{
    puts("Press <Enter>:");               // Нажмите <Enter>
    getchar();
    putchar('H');
    putchar('e');
    putchar('l');
    putchar('l');
    putchar('o');
    putchar('!');
    putchar('\n');
    return(0);
}
```

Введите этот исходный текст и назовите файл `PUTHELLO.C`. Перепроверьте все и исправьте все ошибки. Скомпилируйте и запустите на выполнение:

Press <Enter>.

(Нажмите <Enter>.)

Так и сделайте. Затем увидите:

Hello!

(Привет!)

Обратите внимание, что `getchar()` используется здесь просто для того, чтобы прочесть клавишу <Enter> (ведь только ее вы и нажали); значение, возвращенное `getchar()`, в данной программе не сохраняется. В этом формате `getchar()` все равно возвращает значение, но так как оно не сохраняется в переменной, значение “теряется”. Это вполне допустимо.

После вызова `getchar()` в программе для отображения Hello! (Привет!) и символа новой строки используется семь вызовов функции `putchar()`. Это довольно глупое применение `putchar()`, но оно работает.

✓ Функция `putchar()` отображает один символ на экране.

✓ Для `putchar()` символ можно определить также как escape-последовательность или указать значение кода (см. следующий раздел).

Значения символьных переменных



Если хотите, можете считать, что ключевое слово `char` отводит память для односимвольных переменных и строк. Победа на викторине гарантирована и вы можете прекратить читать книгу...

Ужасная правда состоит в том, что односимвольная переменная на самом деле хранит целое число определенного типа. Крошечное целое число, но, тем не менее, целое число. Это не очевидно. И вот почему. В действительности односимвольные переменные не предназна-

лись изначально для обработки символов (значений типа `char`) как целых чисел. Изначальное предназначение односимвольных переменных состояло в том, чтобы они хранили символы. Но они могут использоваться и как целые числа. Это очень запутанно, так позвольте же мне объяснить все подробно.

В компьютере основная единица памяти (единица хранения) — байт. В компьютере очень много байтов (или мегабайтов) памяти, жесткий диск хранит очень много гигабайтов и т.д. Каждый из байтов можно представить как хранимый в нем единственный символ (единица информации). Байт — символ.

Опуская скучные подробности, я скажу лишь, что в байте можно хранить значения от 0 до 255. Это — диапазон целых чисел типа `unsigned char` (символ без знака): от 0 до 255 (см. табл. 9.1 в главе 9 “Числа в языке C”). Поскольку символ — это байт, символ также может использоваться для хранения таких крошечных целочисленных значений.

Когда компьютер обрабатывает символы, он в действительности не отличает А от В. Однако он на самом деле знает различие между 65 и 66. Внутри компьютера число 65 используется как код, представляющий символ А. Символ В представляет код 66. Фактически все символы алфавита (буквы), символы цифр и другие символы вроде точек, запятых и т.д. имеют свои собственные символьные коды. Схема кодирования называется ASCII, а список кодов и символов находится в приложении Б “Таблица ASCII”.

По существу, когда вы сохраняете символ А в символьной переменной (типа `char`), вы помещаете значение 65 в эту переменную. Внутри компьютер видит только 65 и этим счастлив. Только когда символ “отображается” внешним устройством, оно нам показывает А. Это удовлетворяет вас и меня, если А — то, что мы хотим увидеть.

По этой причине символьные переменные являются и целыми числами, и символами. Правда, они — целые числа. Однако они обрабатываются подобно символам. Следующая программа `WHICH.C` читает символ с клавиатуры и отображает его, используя функцию `printf()`. Уловка в `WHICH.C` состоит в том, что символ отображается и как символ, и как число, т.е. как целочисленное значение. Такая двойственность! Могли бы вы придумать такое?

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

```
int main()                                // главная функция
{
    char key;

    // Нажмите клавишу на клавиатуре
    printf("Press a key on your keyboard:");
    key=getchar();
    // ("Вы нажали '%c' клавишу\n", клавиша)
    printf("You pressed the '%c' key.\n",key);
    // ("Ее ASCII-значение равно %d.\n", клавиша)
    printf("Its ASCII value is %d.\n",key);
    return(0);
}
```

Сохраните файл на диске под именем `WHICH.C`.

Скомпилируйте `WHICH.C`. Если появятся сообщения об ошибках, перепроверьте исходный текст, исправьте его и перетранслируйте.

Выполните полученную программу. Если вы нажмете клавишу <A> (и затем <Enter>), на вашем экране будет отображено вот что:

```
Press a key on your keyboard: A
You pressed the 'A' key.
Its ASCII value is 65.
```

(Нажмите клавишу на клавиатуре: A
Вы нажали клавишу A.
Ее ASCII-значение равно 65.)

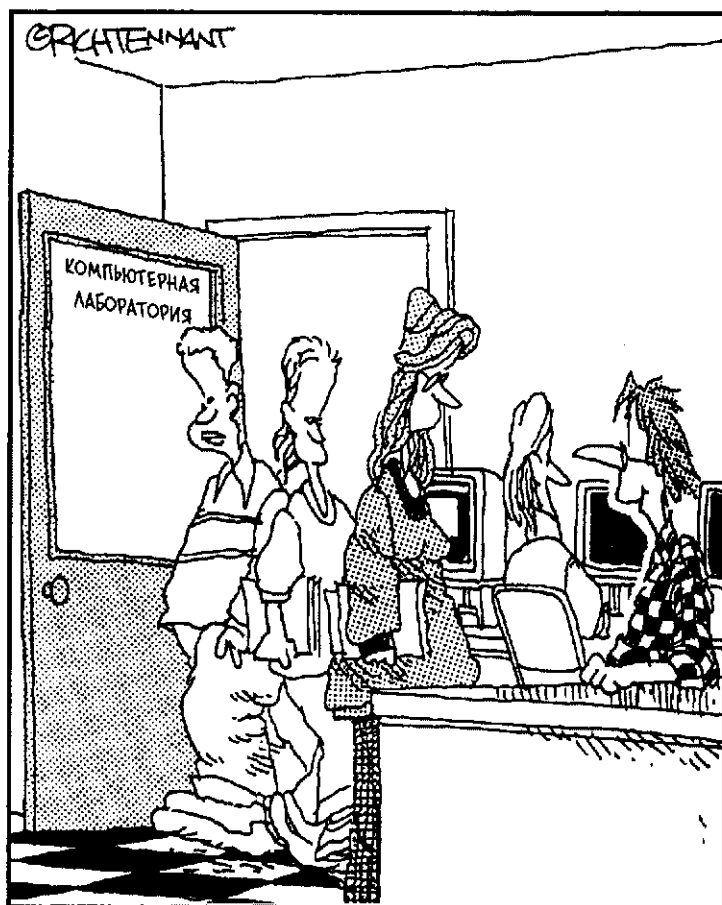
Вторая инструкция `printf()` отображает переменную `key` на месте метки-заполнителя `%s`. Эта метка-заполнитель указывает, что `printf()` должна отобразить переменную как символ. В третьей инструкции `printf()` переменная отображается на месте метки-заполнителя `%d`. Эта метка-заполнитель указывает, что `printf()` должна отобразить переменную как целочисленное значение — его мы и видим на экране.



- ✓ Все символы алфавита — буквы, цифры и другие символы, нанесенные на клавиши клавиатуры, — имеют ASCII-коды.
- ✓ Диапазон значений ASCII-кодов — от 0 до 127.
- ✓ Коды, превышающие 127, — от 128 до 255 — называются расширенными ASCII-кодами. Представляемые ими символы нестандартны на всех компьютерах, хотя они довольно хорошо подобраны в среде Windows. Впрочем, не стоит из-за них волноваться, это — только технические детали.
- ✓ В приложении Б “Таблица ASCII” перечислены все символы ASCII и их значения.
- ✓ В главе 13 “Сравнение символов с помощью ключевого слова `if`” рассказывается, как “сравнить” один символ ASCII (например, букву или иной символ) с другим. Сравниваются на самом деле значения (коды символов), а не их начертания.

Часть III

Как научить программы принимать решения



"Ну, когда я закончу институт, для таких, как я, будет много свободных вакансий. Ведь я создал вирус, который начнет распространяться именно в тот год, когда я закончу учебу..."

В этой части...

Программы, приведенные в первых двух частях этой книги, выполнялись последовательно, т.е. они выполнялись строка за строкой, сверху донизу. Они не делали никаких отклонений, и шаблон выполнения их был неизменным — никакого творческого потенциала, никакого выбора. Скучные программы! Но компьютерные программы способны на большее.

Чтобы действительно заставить программу взбеситься, внутри нее нужно поместить решающую машину. Эта решающая машина позволяет программе делать ту или иную вещь на основании сравнений типа «если пользователь напечатает L, то идите влево, и вас съест голодный эльф». Этот выбор делает не компьютер, компьютер глуп. Но это может сделать инструкция с альтернативой, благодаря которой программа способна на большее, чем выполнять одну за другой последовательно записанные команды.

В дополнение к принятию решений компьютер способен делать вещи много раз — и без жалоб! Объедините принятие решений с этой любовью к повторению — и довольно скоро вы получите программы, которые забросят вас в альтернативные вселенные, взяв компьютер под свой контроль! Пришло время научить программы принимать решения!

Больше математики и Священный порядок (старшинство операций)

В этой главе...

- Математические операторы в C
- Приращение переменных
- Порядок (старшинство операций)
- Представляя вас моей дорогой тетушке Салли
- Использование круглых скобок для управления порядком математических действий

O

стерегайтесь! Вас ждет ужасная математическая глава! Караул!

Некоторым людям математика кажется настолько ужасающей, что я удивлен тем, что в Диснейленде нет никакого математического ужастика. Пираты. Призраки. Кричащие куклы. Все это уже не пугает посетителей. Чтобы по-настоящему ужаснуть и взволновать своих всевозрастных посетителей, Диснейленд нуждается в математике. Людвиг фон Дрейк выглядел бы как Селезень, приглашающий искупаться в аквариуме с акулами. Но я отступаю от темы.

На самом деле эта глава совсем не ужасная математическая глава, а просто моя первая лекция по математике — достаточно длинная, почти настолько длинная, чтобы у вас началась головная боль. Не паникуйте! Всю работу делает компьютер. Вы обязаны только указать математические действия в надлежащем порядке, чтобы получился правильный ответ. А если вы сделаете это неправильно, компилятор с языка C сообщит вам об этом, и вы сможете начать заново. Ничего сложного. Никаких взаимных обвинений. Никаких хихиканий со стороны студентов-очкариков, приехавших по обмену из Трансильвании.

Краткий обзор основных математических операторов языка C

В табл. 11.1 приведены основные математические операторы языка C (можете считать их арифметическими операторами — мне все равно). Эти символы и каракули заставляют программу на языке C выполнять все основные математические действия.

Таблица 11.1. Знаки математических операций в языке C

Оператор, или символ (знак) операции	Смысл	Как называется шестиклассниками	Выполняемое действие
+	+	"плюс"	сложение
-	-	"минус"	вычитание
*	×	"умножить"	умножение
/	:	"разделить"	деление

Эти символы позволяют выполнять математические операции следующих типов.

✓ **Выполнение действий непосредственно над значениями:**

`total = 6 + 194;` // общее количество = 6 + 194;

Целочисленная переменная `total` (общее количество) содержит результат сложения 6 и 194.

Рассмотрим несколько примеров.

`result = 67 * 8;`

Переменная `result` (результат), которая может быть целочисленной переменной или переменной с плавающей точкой, содержит результат умножения 67 на 8.

`odds = 45/122;`

Переменная с плавающей точкой `odds` содержит результат деления 45 на 122.

Во всех случаях сначала выполняется математическая операция, указанная справа от знака "=". Математические действия компилятор с языка C выполняет слева направо. Значение, которое является результатом, помещается в числовую переменную.

✓ **Выполнение действий над значениями и переменными:**

`score = points*10;`

Переменная `score` (счет) равна значению переменной `points`, умноженному на 10.

✓ **Выполнять действия можно над чем угодно — над функциями, значениями, переменными или любыми их комбинациями:**

`height_in_cm = atoi(height_in_inches)*2.54;`

Переменная `height_in_cm` равна значению, возвращенному функцией `atoi`, умноженному на 2.54. Функция `atoi()` принимает в качестве параметра переменную `height_in_inches` (значение которой, вероятно, вводится с клавиатуры в виде строки).

✓ **Сначала слева направо выполняются математические действия. Затем результат присваивается переменной, находящейся слева от знака "=".**



Старая программа определения роста

Чтобы сделать некоторые хоть и простые, все же раздражающие математические расчеты, вы можете использовать компьютер. Как пример, я представляю программу `HEIGHT.C` и ее исходный текст. Эта программа просит ввести ваш рост в дюймах, а затем выводит результат в сантиметрах. Это типичная унылая программа на языке C. Но потерпите несколько страниц, и позабавляйтесь с ней. Введите эту тривиальную программу с помощью вашего редактора:

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

```
int main() // главная функция
{
    float height_in_cm;
    char height_in_inches[4];

    // Введите ваш рост в дюймах
    printf("Enter your height in inches:");
    gets(height_in_inches);
    height_in_cm = atoi(height_in_inches)*2.54;
    // Рост в сантиметрах
    printf("You are %.2f centimeters tall.\n",height_in_cm);
    return(0);
}
```

Будьте внимательны: в программе есть переменные с длинными именами. Кроме того, пишете *height* (высота), а не *hieght*. (Я говорю об этом, потому что я дважды пробовал компилировать программу с этой ошибкой правописания — не однажды!) Сохраните файл на диске под именем **HEIGHT.C**.

Скомпилируйте программу. Исправьте все синтаксические и другие серьезные ошибки. Устраните их, если они неожиданно возникнут.

Выполните программу **HEIGHT**. Вывод будет примерно таким:

```
Enter your height in inches:60
You are 152.40 centimeters tall.
```

(Введите ваш рост в дюймах:60
Ваш рост 152.40 сантиметра.)

Если ваш рост равен 60 дюймов (в точности 5 футов), то он равен 152,40 сантиметра — число хотя и большее, но рост-то все равно не изменился. Программа способна преобразовывать почти любую длину в дюймах в ее соответствующее выражение в сантиметрах.

- ✓ *Height* (высота). В этом слове *e* предшествует *i*. Это еще один пример того, почему орфография английского языка считается самой сложной среди всех языков планеты. (Это самая вероятная причина опечаток, если в программе есть синтаксические ошибки.)
- ✓ Функция `atoi()` строку символов (в данной программе введенную пользователем) преобразует в целочисленное значение. И это функция `atoi()`, а не `atio()`. (Еще одна причина ненавидеть правописание английского языка, хотя на этот раз английский язык ни при чем!)
- ✓ Функция `atoi()` переводит значение, хранимое в строковой переменной `height_in_inches`, в целое число. Затем это значение умножается на 2,54. (В качестве знака умножения используется звездочка.) После этого результат перескакивает через знак "=" и сохраняется в переменной типа `float` (с плавающей точкой) `height_in_cm`.
- ✓ Очевидно, значение 2,54 имеет тип `float` (с плавающей точкой), потому что оно представляет собой десятичную дробь. `height_in_inches` — целое число, потому что именно этот тип значения возвращает функция `atoi()`. Когда вы умножаете целое число на число типа `float` результат будет иметь тип `float`. Именно поэтому переменная `height_in_cm` объявлена как имеющая тип `float`.



- ✓ В одном дюйме 2,54 сантиметра. Помнить это совсем не обязательно, во всяком случае, в Соединенных Штатах. Однако иногда это может помочь проявить эрудированность на вечеринке. Вы можете как бы ненароком сказать: “Ваш нос сейчас находится на расстоянии 2,54 сантиметра от салата, а это, знаете ли, как раз 1 дюйм”. (Между прочим, запомнить, сколько сантиметров в дюйме, совсем просто: нужно 30 раз прокричать “два-точка-пять-четыре сантиметра в дюйме” во всю силу ваших легких.)
- ✓ Сантиметр равен 0,39 дюйма. Сантиметр примерно равен толщине большого пальца, если, конечно, вы не забиваете им гвозди.

Изменение старой программы определения роста

Загрузите исходный текст программы HEIGHT.C в ваш редактор. Хорошо. Измените строку `height_in_cm = atoi(height_in_inches)*2.54;` следующим образом:
`height_in_cm = atoi(height_in_inches)*2.54*1.05;`

Иными словами, после 2.54 и перед точкой с запятой вставьте ***1.05** (“умножить на один точка ноль пять”). Это увеличивает ваш рост на пять сотых сантиметра для каждого сантиметра. Результат? Ждите! Сохраните файл и перетранслируйте его. Затем выполните его снова:

```
Enter your height in inches:60
You are 160.02 centimeters tall.
```

(Введите ваш рост в дюймах:60
Ваш рост 160,02 сантиметра.)

Это совсем немного. Но если предположить, что вы переписываетесь с некоторой француженкой (или французом), которая (который) романтично интересуется Вами... Если так, вы можете сказать ему или ей, что, согласно программе, разработанной на вашем компьютере, ваш рост составляет 160,02 сантиметра. Это ничего не означает для американцев, но во Франции будут думать, что вы на целых три дюйма выше. Если бы ваш рост составлял 70 дюймов, программа выдала бы следующее:

```
Enter your height in inches:70
You are 186.69 centimeters tall.
```

(Введите ваш рост в дюймах:70
Ваш рост 186,69 сантиметра.)

Теперь ваш рост 186,69 сантиметра! Французы и француженки будут в обмороке!
А теперь признание.

Цель этого обсуждения состоит не в том, чтобы рассказать вам, как сделать компьютерные программы соучастниками обмана, и уж совсем не в том, чтобы обмануть французов. В большинстве случаев люди, которые выполняют программы, хотят получить точные результаты. Однако в действительности переменная `height_in_cm` равна результату трех математических операций. И в самом деле,

```
height_in_cm = atoi(height_in_inches)*2.54*1.05;
```

Сначала значение (целое число) получается в результате преобразования строковой переменной `height_in_inches`. Полученное значение умножается на 2.54, а затем результат снова умножается на 1.05.

Длинная математическая формула вполне допустима в программе на C. Вы можете все время складывать, умножать, делить и делать всякую всячину. Чтобы гарантировать, что вы всегда получаете нужный вам результат, необходимо обратить особое внимание на то, что называется порядком выполнения действий или старшинством операций. Это тема следующего раздела в этой главе.



- ✓ В одной инструкции С можно выполнять действия над несколькими, не обязательно над двумя операндами. На самом деле в строке (инструкции) можно выполнить несколько операций над самыми разнообразными операндами. Конечно, все операнды должны быть справа от знака “=”.
- ✓ Чтобы увеличить значение на .05 (пять сотых, или 5 процентов), его нужно умножить на 1.05. Если вы умножаете значение только на .05, вы уменьшаете его на 95 процентов. Но если вы хотите увеличить его на 5 процентов, умножьте его на 1.05. Между прочим, я узнал это случайно.

Тонкое искусство приращения (добавления единицы)

Математическая операция добавления единицы в программировании на С называется приращением, или инкрементированием. Вы увеличиваете что-нибудь — например, переходя от 1 к 2 или увеличивая вашу компенсацию на один доллар в час. Это примеры приращения.

Увеличение значения переменной в С встречается очень часто. Оно выполняется с помощью следующего пугающего равенства:

```
i=i+1;
```

Это равенство имеет единственную цель: добавить 1 к значению переменной *i*. Оно выглядит забавно, но работает.

Предположим, что *i* равно 3. Тогда *i*+1 (это 3 + 1) равняется 4. Поскольку в С сначала вычисляется правая часть (то, что стоит справа от знака “=”), значение 4 перескакивает через знак “=” и помещается в переменную *i*. Предыдущая инструкция *увеличивает* значение переменной *i* на 1. Иными словами, выполняется операция *приращения*, или *инкремента*.

Тем же способом можно добавить к значению не только 1, но и другое число. Например:

```
i=i+6;
```

Это равенство увеличивает (инкрементирует) значение переменной *i* на 6. (Конечно, юристы могут сказать, что слово *инкрементирует* означает “добавить 1”. Но что они понимают?)

- ✓ Чтобы добавить 1 к переменной *i*, используйте следующую (математическую) инструкцию языка С:

```
i=i+1;
```

Это называется *приращением* или *инкрементацией*.

- ✓ Нет, это не *инкриминирование* (изобличение). Совершенно разные вещи.
- ✓ Примеры приращения значений могут быть самыми разными: увеличение высоты самолета (или космического корабля); увеличение возраста после дня рождения; приращение номера рыбы, съеденной вашим котом; увеличение веса за отпуск.
- ✓ Приращение *i=i+1* работает, потому что в С сначала вычисляется то, что находится справа от знака “=”. Так что сначала вычисляется *i*+1. Затем полученное значение заменяет первоначальное значение переменной *i*. Все это может вскружить вам голову, только если вы постоянно мотаете головой слева направо.

Приращение веса — к ожирению

Следующая программа — LARDO.C — довольно грубая интерактивная программа, в которой математические действия используются для того, чтобы увеличить ваш вес. Вы вводите ваш вес, а затем LARDO вычисляет ваш вес после банкета (праздника, отпуска...):

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

int main()                                // главная функция
(
    char weight[4];
    int w;

    // Введите ваш вес
    printf("Enter your weight:");
    gets(weight);                          // вес
    w=atoi(weight);

    // Пока вы весите
    printf("Here is what you weigh now: %d\n",w);
    w=w+1;

    // Ваш вес после картофеля
    printf("Your weight after the potatoes: %d\n",w);
    w=w+1;

    // Вот, пожалуйста, после баранины
    printf("Here you are after the mutton: %d\n",w);
    w=w+8;

    // И ваш вес после десерта
    printf("And your weight after dessert: %d pounds!\n",w);
    printf("Lardo!\n");
    return(0);
)
```

Введите предыдущий исходный текст с помощью вашего текстового редактора. По-настоящему новой для вас в этом примере может быть только инструкция `w=w+1`, которая увеличивает значение переменной `w` на единицу. Последняя инструкция, `w=w+8`, добавляет восемь к значению переменной `w`.

Проверьте введенный текст (не забудьте проверить также точки с запятой и двойные кавычки). Сохраните файл на диске под именем LARDO.C.

Скомпилируйте LARDO.C. В случае необходимости исправьте все ошибки.

Следующая выдача была получена в результате прогона данной программы, причем в качестве веса пользователя было введено 175:

```
Enter your weight:175
Here is what you weigh now: 175
Your weight after the potatoes: 176
Here you are after the mutton: 177
And your weight after dessert: 185 pounds!
Lardo!
```

```
(Введите ваш вес:175
Пока вы весите: 175
Ваш вес после картофеля: 176
Вот, пожалуйста, после баранины: 177
И ваш вес после десерта: 185 фунтов!)
```

- ✓ Программы не должны оскорблять пользователей! Идея этого примера состоит в том, чтобы показать, как используется инструкция `w=w+1` для добавления 1 к значению переменной. Добавление 1 называется приращением. (Это то, что Бог делает с вашим весом каждое утро, чтобы вы не пришли на работу преждевременно.)
- ✓ Да, 175 фунтов! Я уверен, что вы напечатали это скромное значение, а не кое-что более правдоподобное, учитывая ваш представительный вид и, в особенности, вашу талию.

Еще одна программа! (Она может даже понадобиться в жизни)

Возможно, Монополия — одна из самых великолепных когда-либо изобретенных настольных игр, и это может быть потрясающая забава — особенно когда вы обладаете массивами гостиниц, а ваши жалкие противники поселяются в них подобно тому, как глупые мухи весь день садятся на липучку. Единственная проблема в этом случае состоит в том, чтобы не вытянуть карту уплаты в общий сундук (Community Chest), которая гласит следующее: “На ремонт улиц вы должны уплатить по 40\$ за здание, по 115\$ за гостиницу”.

Вы подсчитываете все ваши здания и умножаете их количество на 40\$, а количество гостиниц — на 115\$ (согласитесь, очень странное число); затем вы складываете оба значения. Это ужасно трудно, особенно в процессе игры, но тяжелую умственную работу может облегчить простая компьютерная программа, например ASSESSED.C:

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

int main()                                // главная функция
{
    // здания, гостиницы, общее количество
    int houses, hotels, total;
    char temp[4];

    // Введите количество зданий
    printf("Enter the number of houses:");
    gets(temp);
    houses=atoi(temp);

    // Введите количество гостиниц
    printf("Enter the number of hotels:");
    gets(temp);
    hotels=atoi(temp);

    total=houses*40+hotels*115;

    // ("Вы должны банку $%d.\n ", общее количество)
    printf("You owe the bank $%d.\n",total);
    return(0);
}
```

Тщательно напечатайте эту программу с помощью вашего редактора в новом документе. Перепроверьте точки с запятой, круглые скобки и кавычки. Затем сохраните файл на диске под именем ASSESSED.C.

Скомпилируйте! Исправьте все ошибки (в случае необходимости). Затем выполните программу. Предположим, что вы имеете девять зданий и три гостиницы. Тогда вывод будет таким:

```
Enter the number of houses:9
Enter the number of hotels:3
You owe the bank $705.
```

```
(Введите количество зданий:9
Введите количество гостиниц:3
Вы должны банку 705$.)
```

Удивительно, как легко компьютер смог понять это! Конечно, в такой ситуации вы легко можете позволить себе 705\$ игровых (а не реальных) денег. Все, что вам нужно — немножко сока и номер в гостинице на площади Святого Чарльза, и вы мигом вернете свои деньги обратно.



- ✓ Заметьте, как используется временная переменная `temp` для хранения и преобразования двух различных строк в числа. Этот пример иллюстрирует, как могут изменяться переменные.
- ✓ Математические вычисления в инструкции `total=houses*40+hotels*115`; выполняются правильно, потому что компилятор принимает во внимание то, что в следующем разделе называется Священным порядком, или старшинством операций.
- ✓ Вы можете подумать (и это действительно так), что общее количество `total`, отображаемое программой, должно быть переменной типа `float` (с плавающей точкой). В конце концов, суммы долларов обычно имеют десятичную дробную часть: 705,00\$, а не 705\$. Но в данном случае все значения — целые числа, и потому лучше значение общего количества хранить в целочисленной переменной `total`. Имейте в виду, операции над целыми числами выполняются быстрее; правда, это особенно очевидно лишь в больших программах.

Священный порядок, или старшинство операций

Старшинство определяет, что выполняется сначала. Пожар в театре, например, значительно важнее того, что вы можете пропустить второе действие, если уделите второпях.

Порядок, или старшинство указывает приоритет математических операторов. Например, знак “плюс” можно вставить между числами, но это далеко не всегда приведет к сложению тех чисел, между которыми он стоит. В С перед сложением выполняются другие математические операции. И это правильно.

Задача из выпускного экзамена по лечению зубов

Взгляните на следующее длинное и сложное математическое равенство, которое может однажды проникнуть в одну из ваших программ на С:

```
answer = 100 + 5 * 2 - 60 / 3;           // ответ
```

Этот вопрос — один из самых трудных математических вопросов на выпускном экзамене по лечению зубов. Да, большинство дантистов предпочитает выдернуть зубы — даже их собственные, чем вычислить это выражение. Но для вас это не проблема. Ответ вычислит компьютер. Но что это?

Ответ — 50? Сто плюс 5 — 105; умножаем на 2 — 210; минус 60 — 150; делим на 3, получаем 50. Компилятор заставит компьютер вычислить это выражение для вас автоматически? Почему в переменную `answer` (ответ) помещено значение 90?

Девяносто? Да, значение переменной `answer` — 90. Это все имеет отношение к Моей Дорогой Тетушке Салли¹ и порядку (старшинству) операций. Перед обсуждением подробностей введите следующую программу `DENTIST.C` и убедитесь, что переменная `answer` равна 90, а не 50:

¹ Тетушка Салли (Aunt Sally) — непрменный персонаж всех детских учебников и задачников по арифметике в США. — Прим. ред.

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

```
int main()
```

```
// главная функция
```

```
{  
    printf("%d", 100+5*2-60/3);  
    return(0);  
}
```

Введите эту короткую замечательную программу с помощью вашего редактора. Скомпилируйте ее. Выполните. Главное в этой программе — простая инструкция `printf()`, причем в двойных кавычках указан только символ преобразования `%d`. После строки идет запятая, а затем — математическое выражение, которое требовалось вычислить на выпускном экзамене по лечению зубов.

Выполните программу. Результат должен потрясти вас:

90



- ✓ Порядок математических операций очень важен. Если вы не знаете порядок выполнения математических операций компилятором с языка C, вы не можете получить правильный ответ. Именно поэтому вы должны знать порядок (старшинство) математических операций и, что еще более важно, никогда не перечить Моей Дорогой Тетушке Салли.
- ✓ Когда выполняется программа DENTIST.C, сначала компьютер вычисляет выражение $100+5*2-60/3$ в функции `printf()`. Затем результат вставляется на место заполнителя `%d` и отображается на экране.
- ✓ Я мог бы написать программу DENTIST.C так, чтобы объявить целочисленную переменную `answer`, присвоить значение этой переменной, а затем с помощью `printf()` отобразить содержимое этой переменной. Но это было бы слишком длинно для такой программы. В языке C можно писать короткие программы. Инструкция `printf()` в DENTIST.C — только один из примеров сокращения программ на C.

Что же не так, Моя Дорогая Тетушка Салли?

В данном случае Моя Дорогая Тетушка Салли (My Dear Aunt Sally) — всего лишь глупая мнемоническая фраза, позволяющая запомнить порядок действий:

Умножение (Multiplication)

Деление (Division)

Сложение (Addition)

Вычитание (Subtraction)

MDAS — порядок, в котором выполняются математические действия в длинных математических выражениях на языке C. Это и есть порядок (старшинство) математических операций.

Компилятор просматривает все выражение — всю строку — и сначала выполняет умножение и затем деление, а только после этого — сложение и вычитание. Так что не все действия выполняются тупо слева направо. На рис. 11.1 показано, как вычисляется математическое выражение из примера в предыдущем разделе. Как видите, результат действительно равен 90.

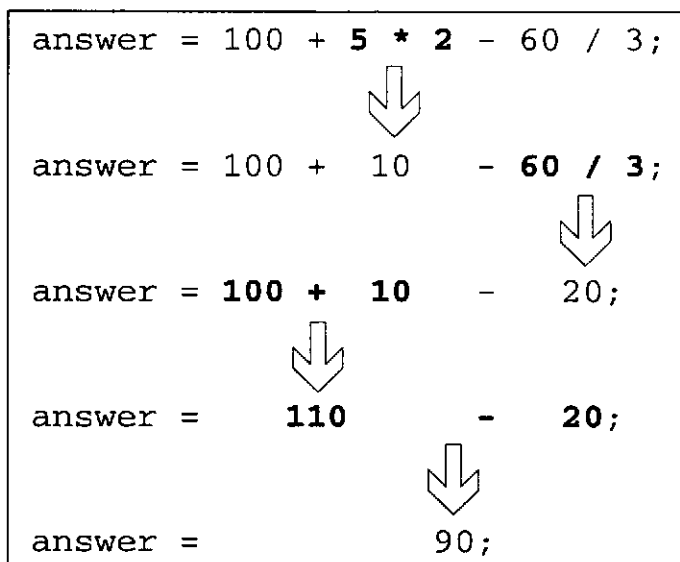


Рис. 11.1. Вот так компилятор с языка C вычисляет длинное математическое выражение; результат записывается в переменную *answer* (ответ)

Вот еще одна задача:

`answer = 10 + 20 * 30 / 40;`

В этой инструкции сначала выполняется умножение, а затем деление и только после него — сложение. Когда умножение и деление стоят рядом друг с другом, как в предыдущей строке, они выполняются слева направо.

Когда компьютер закончит выполнение предыдущей инструкции, переменная *answer* будет содержать значение 25. Вы можете доказать это, отредактировав программу DENTIST.C и заменив в ней математическое выражение, которое уже есть там, новым математическим выражением. Перетранслируйте и выполните программу, чтобы убедиться, что ответ равен 25. Или доверьтесь мне и читайте дальше.



- ✓ *Моя Дорогая Тетушка Салли — My Dear Aunt Sally.* Multiplication — умножение (*), division — деление (/), addition — сложение (+) и subtraction — вычитание (–) — именно в этом порядке выполняются действия в длинных математических выражениях языка C.
- ✓ Порядок (старшинство) операций важен потому, что вы должны правильно записать математические выражения, чтобы получить правильный ответ.
- ✓ В программе ASSESSED.C из предыдущего раздела также используется порядок (старшинство) операций:
`total=houses*40+hotels*115;`
 Сначала количество зданий (*houses*) умножается на 40, а затем количество гостиниц (*hotels*) — на 115. Последнее действие — сложение полученных двух произведений.
- ✓ Чтобы изменить порядок (старшинство) операций, используются круглые скобки. Это обсуждается далее в этой главе в разделе “Круглые скобки могут изменить приоритет операций, определяемый их порядком (старшинством)”.

Арифметическая путаница с волшебными шариками

Я ненавидел все математические текстовые задачи, когда был ребенком. Фактически, я все еще ненавижу их и сейчас. Но как бы то ни было, даже в моей взрослой жизни мне приходится решать задачи, подобные следующей.

Предположим, что вы имеете 100 волшебных шариков. Их количество удваивается в каждые 24 часа. Через день их у вас было бы 200. Но сначала вы должны дать мяснику 25 волшебных шариков в обмен на глазные яблоки трех дюжин ягнят для блюда, которым вы хотите удивить вашего супруга (супругу). Так сколько же волшебных шариков будет у вас на следующий день?

Давайте напишем выражение

```
100 - 25 * 2;
```

Это — 100 волшебных шариков минус 25 для глазных яблок и затем умножаем на 2 (удваиваем), чтобы подсчитать их количество на следующий день. Вы можете полагать, что 100 минус 25 равно 75. Умножьте 75 на 2, и у вас получится 150 волшебных шариков на следующий день. Но в C соблюдается порядок (старшинство) операций (установленный этой строгой тетушкой Салли), и потому сначала 25 умножилось бы на 2. В результате вычисления написанного выше выражения получилось бы 50 волшебных шариков на следующий день. Какой обман!

Следующая программа на C, PELLETS.C, иллюстрирует, как порядок (старшинство) операций путает решение задачи о волшебных шариках. Эта программа — несколько более сложная версия основной программы DENTIST.C, представленной ранее в этой же главе.

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

```
int main() // главная функция
{
    int total; // общее количество

    total=100-25*2;
    // ("Завтра у вас будет %d волшебных шариков.\n ", общее количество)
    printf("Tomorrow you will have %d magic pellets.\n",total);
    return(0);
}
```

Введите эту программу с помощью вашего редактора. Перепроверьте все. Сохраните файл на диске под именем PELLETS.C.

Скомпилируйте PELLETS.C. Исправьте все ошибки.

Выполните программу PELLETS. Вот что получится:

```
Tomorrow you will have 50 magic pellets.
```

(Завтра вы будете иметь 50 волшебных шариков.)

Угу. Попробуйте объяснить это налоговой службе. Ваша компьютерная программа, старательно введенная, говорит вам, что завтра вы будете иметь только 50 шариков, хотя в действительности — 150. А куда девались мои 100 кровных? Они были потеряны из-за порядка (старшинства) операций. В программе PELLETS.C сначала нужно выполнить сложение (с отрицательным числом — тем, которое выражает количество шариков, отданных мною мяснику), или вычитание. Для этого нужно изменить приоритет операций с помощью круглых скобок.

Круглые скобки могут изменить приоритет операций, определяемый их порядком (старшинством)

Моя Дорогая Тетя Салли может быть весьма властной. Она настойчива. Однако даже при том, что обычно она все устраивает хорошо, иногда она попадает впросак. В программе PELLETS.C например, она говорит компилятору языка C, что сначала он должен умножить

25 на 2, а затем вычесть полученный результат из 100. Но даже самый последний двоечник, прочитав условие задачи, скажет, что сначала вы должны вычесть 25 из 100, а затем умножить результат на 2. Проблема теперь состоит в том, чтобы убедить в этом компилятор языка С (и Тетушку Салли). Как же это сделать?



Чтобы изменить приоритет операций, заданный их порядком (старшинством), используются круглые скобки. Когда компилятор языка С видит круглые скобки, он быстро вычисляет математическое выражение, заключенное в них, а затем продолжает вычислять слева направо оставшееся вне круглых скобок выражение, учитывая порядок (старшинство) умножения, деления, сложения и вычитания.

Чтобы исправить программу PELLETS.C, нужно изменить выражение так:

```
total=(100-25)*2;
```

```
// общее количество = (100-25) *2;
```

Компилятор языка С сначала вычисляет математическое выражение в круглых скобках. Поэтому сначала 25 вычитается из 100, результат равняется 75. Затем вычисляется остальная часть математического выражения: 75 умножается на 2 — получается 150. Это и есть правильный ответ в задаче о волшебных шариках.

Я прошу вас сделать указанное изменение выражения в программе PELLETS.C. Вставьте левую круглую скобку перед 100, а правую — после 25. Сохраните измененный файл на диске, перетранслируйте его, а затем выполните программу. Результат должен порадовать вас:

Tomorrow you will have 150 magic pellets.

(Завтра вы будете иметь 150 волшебных шариков.)

- ✓ Всегда сначала вычисляются математические выражения, заключенные в круглые скобки. Не имеет значения, какая операция выполняется — сложение или вычитание, но именно она всегда выполняется в выражении сначала.
- ✓ Математическое выражение в круглых скобках вычисляется слева направо. Кроме того, в круглых скобках приоритет все равно имеют умножение и деление. Однако независимо от того, что находится в круглых скобках, оно вычисляется раньше того, что находится снаружи. Вот краткое резюме:
 1. Сначала выполняются все операции в круглых скобках.
 2. Сначала выполняются умножение и деление, а затем сложение и вычитание.
 3. Вычисления выполняются слева направо.
- ✓ Если вы когда-либо работали со сложными уравнениями электронной таблицы, вы знакомы с применением круглых скобок: они могут использоваться для изменения порядка выполнения математических операций. Если же вы не используете электронные таблицы, тогда читайте об этом в книге по С, и вы научитесь применять круглые скобки при работе с электронными таблицами.
- ✓ Иногда круглые скобки даже помещают внутри других круглых скобок. Но в этих случаях удостоверьтесь, что левая скобка соответствует правой; круглые скобки жуликоваты — они могут порождать синтаксические ошибки, точно так же, как это происходит при отсутствии двойных кавычек и фигурных скобок.
- ✓ Не имеет значения, где круглые скобки находятся в математическом выражении; то, что находится в них, всегда вычисляется сначала. Например:

```
total=2*(100-25);
```



В этой инструкции сначала вычисляется 100 минус 25. Затем результат 75 умножается на 2. Это правило справедливо независимо от того, насколько сложным является выражение.

Могущественная команда if

В этой главе...

- Использование условного оператора
- Сравнение значений с помощью команды if
- Форматирование условных операторов if
- Обработка исключительных ситуаций с помощью else
- Принятие множественных альтернативных решений

Хорошо, if — не команда. Это просто еще одно ключевое слово в языке программирования C. Его вы можете использовать в вашей программе, чтобы принять решение — хотя на самом деле оно выполняет операцию сравнения, а ничего не решает. Именно программа решает, что сделать. Правда, свое решение она принимает на основании результатов операций сравнений.

Эта глава о том, как принимать решения в программах с помощью команды if.



Имейте в виду, что компьютер не решает, что сделать. Вместо этого он следует инструкциям, написанным для этого программистом. Это похоже на инструкции для маленьких детей, когда вы хотите, чтобы они сделали что-то, хотя компьютер отличается тем, что всегда делает в точности то, что вы ему говорите, и никогда не отвлекается, уставившись в телевизор или укладывая свою любимую кошку на только что убраный диван.

Если бы только знать, как использовать if...

Идея команды if состоит в том, чтобы заставить компьютер обрабатывать некоторые предсказуемые и все же неизвестные (случайные) события: выбор, сделанный с помощью меню; события в некоторой игре; или нечто, введенное пользователем. Это все примеры событий, которые могут произойти, и компьютер должен уметь обрабатывать их.

Ключевое слово if позволяет помещать типичные случаи принятия решений в программы. Решения принимаются на основе результатов операций сравнения. Например:

- ✓ Если содержимое переменной *X* больше, чем содержимое переменной *Y*, крикни так, вроде тебя укусил за нос самый большой марсианский дракон.
- ✓ Если содержимое переменной *calories* (калории) превышает допустимый предел, блюдо, должно быть, очень вкусное.

- ✓ Если вещь не сломалась, ее не надо чинить.
- ✓ Если Дуг не пригласит меня на школьный бал, придется выйти замуж за Чарли.

Все эти примеры показывают важность принятия решений; подобные решения вы можете принимать в ваших программах на С с помощью ключевого слова `if`. Однако в языке программирования С операции сравнения, выполняемые с помощью ключевого слова `if`, имеют математический характер. Вот более конкретные примеры условий:

- ✓ если значение переменной `A` равно значению переменной `B`;
- ✓ если содержимое переменной `ch` меньше чем 132;
- ✓ если значение переменной `zed` (название буквы Z) больше чем 1 000 000.

Эти примеры условий действительно просты, все вычисления в них производятся над переменными и значениями. Ключевое слово `if` заставляет компьютер использовать результат операции сравнения, и если результат операции сравнения — истина (`true`), программа выполняет указанную инструкцию.

- ✓ `if` — ключевое слово в языке программирования С. Оно позволяет программам принимать решения.
- ✓ `if` решает, что сделать, на основе операции сравнения, примененной (обычно) к двум выражениям.
- ✓ Неравенство, которое используется в `if`, имеет математический характер: два выражения могут быть равны, первое из них может быть больше, чем второе, первое может быть меньше, чем второе, и так далее. Если они равны, выполняется определенная часть программы. В противном случае эта часть программы не выполняется.
- ✓ С ключевого слова `if` начинается то, что называется инструкцией выбора в языке С. Инструкция выбора. Произведите впечатление на ваших друзей этим термином, если вы можете запомнить его. А если они спросят, что это такое, просто задержите свой нос. (Именно так делаю я.)



Пример глупой компьютерной программы

Следующая программа — `GENIE1.C`, одна из многих тех самых глупых компьютерных программ типа “угадай число”, которые пишут студенты, когда учатся программировать. Когда компьютеры еще только появились, компьютерщики просиживали над этими играми в течение многих часов. Они, вероятно, упали бы замертво, если бы увидели сияние Sony PlayStation в те далекие годы.

`GENIE1.C` просит задумать целое число от 0 до 9. Вы вводите задуманное число с клавиатуры. Затем, используя волшебство условного оператора, компьютер говорит вам, меньше ли пяти введенное вами число. Впервые такая программа была написана в начале 1950-х. Это был главный хит в те годы.

Введите следующий исходный текст с помощью вашего текстового редактора. Новым для вас будет только условный оператор почти в самом конце программы. Лучше дважды перепроверьте введенный текст.

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

```
int main() // главная функция
{
    char num[2];
    int number;
```

```
// Я волшебная компьютерная программа!
printf("I am your computer genie!\n");
// Введите число от 0 до 9:
printf("Enter a number from 0 to 9:");
gets(num); // цифра
number=atoi(num);
if(number<5) // если (цифра <5)
{
    // Это число меньше, чем 5!
    printf("That number is less than 5!\n");
}
// Волшебная программа знает все, видит все!
printf("The genie knows all, sees all!\n");
return(0);
}
```

Сохраните файл на диске под именем GENIE1.C.

Скомпилируйте GENIE1.C. Если будут ошибки, с помощью редактора исправьте их. Затем перетранслируйте.

Выполните полученную программу. На экране вы увидите:

```
I am your computer genie!
Enter a number from 0 to 9:
```

(Я волшебная компьютерная программа!

Введите число от 0 до 9:)

Введите число в диапазоне от 0 до 9. Например, вы можете напечатать 3. Нажмите <Enter>, и вы увидите:

```
That number is less than 5!
The genie knows all, sees all!
```

(Это число меньше, чем 5!

Волшебная программа знает все, видит все!)

- ✓ Директива `#include <stdlib.h>` необходима, потому что в программе используется функция `atoi()`.
- ✓ После команды `if` следуют круглые скобки, в которых находится операция сравнения, результат этой операции проверяет ключевое слово `if`.
- ✓ Операция сравнения, результат которой проверяет ключевое слово `if`, состоит в том, что значение переменной `number` (число) сравнивается с 5. Символ `<` между ними означает “меньше чем”. Операция сравнения читается: “Если значение числовой переменной `number` меньше, чем 5”. Если это высказывание истинно, выполняются инструкции после ключевого слова `if`. Если высказывание оказывается ложным, инструкции после ключевого слова `if` пропускаются.
- ✓ Вы хоть помните из школы, что `<` — меньше чем? Хорошо!
- ✓ Обратите внимание, что после проверки, выполняемой ключевым словом `if`, не ставится точка с запятой! Вместо точки с запятой там стоит инструкция, заключенная в фигурные скобки. Инструкции (а их может быть несколько) “принадлежат” команде `if` и выполняются, только если условие (проверяемое ключевым словом `if`) истинно.
- ✓ Если вы видите только строку `The genie knows all, sees all!` (Волшебная программа знает все, видит все!), вы, вероятно, напечатали число большее, чем 4 (например, 5 или больше). Причина состоит в том, что условный оператор проверяет, меньше ли значение, чем 5, и только это условие. Если значение меньше, чем 5, отображается `That number is less than 5!` (Это число меньше, чем 5!). В следующем разделе уточняется, как все это работает.

- ✓ Нет, волшебная компьютерная программа ничего не знает и ничего не видит, если вы ввели 5 или большее число.
- ✓ Обратите внимание на фигурные скобки в середине этой программы. Это — часть условного оператора. Обратите внимание также на то, как они выровнены.

Ключевое слово if

Это слово отличается от любого другого ранее виденного вами слова языка C. Ключевое слово `if` имеет уникальный формат, с множеством вариантов и местом для всяких глупостей. Все же это удобная и мощная инструкция, не бойтесь вставлять ее в программы. И вообще, она используется очень часто.

Ключевое слово `if` используется, чтобы принять решение в программе. Оно проверяет результат операции сравнения. Если результат равен `true` (истина), выполняется остальная часть условного оператора. Если результат операции сравнения не является истиной, программа перекакивает через остальную часть условного оператора, как показано на рис. 12.1.

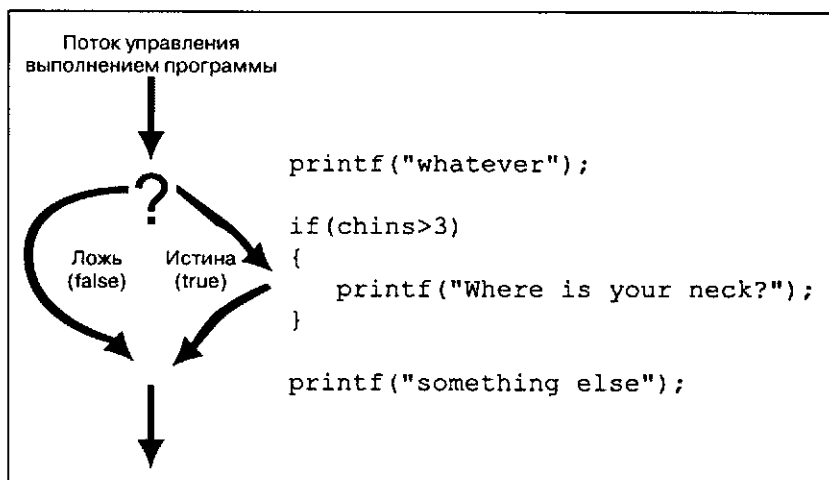


Рис. 12.1. Так выполняется инструкция `if`

Условный оператор представляет собой операторный “блок”, который может выглядеть следующим образом:

```

if (неравенство)
{
    инструкция;
    [инструкция; ...]
}
    
```

После ключевого слова `if` следует пара круглых скобок, в которых записано то, что мы будем называть “неравенство”¹. Неравенство имеет математический характер; знаки операций сравнения приведены в табл. 12.1. Обычно сравнивается значение переменной с постоян-

¹ В оригинале *comparision* (сравнение). Все же я перевожусь это слово как неравенство, чтобы не спутать его со сравнением по модулю. То, что стоит в круглых скобках условного оператора, не обязано быть неравенством в привычном “школьном” смысле, хотя чаще всего именно им оно и является. По этой причине то, что стоит в круглых скобках условного оператора, часто называется условием. Впрочем, условием иногда называется и вся конструкция — ключевое слово `if`, его круглые скобки и то, что стоит в них. — Прим. перев.

ным значением или две переменные друг с другом (см. примеры в табл. 12.1). Если результат операции сравнения истинен, выполняется инструкция (или группа инструкций) между фигурными скобками. Если результат является ложным, программа перескакивает через инструкции в фигурных скобках — программа попросту их игнорирует.

Да, фигурные скобки, которые следуют после ключевого слова `if`, могут содержать несколько инструкций, а не только одну. И каждая из этих инструкций заканчивается точкой с запятой. Все они заключены в фигурные скобки. Это называется блоком кода. Он указывает, какие инструкции “принадлежат” ключевому слову `if`. И все это является частью условного оператора.

Таблица 12.1. Знаки операций сравнения, используемые в условных операторах `if`

Знак операции сравнения	Значение или чтение	Примеры истинных неравенств
<	меньше чем	1 < 5 8 < 9
==	равно	5 == 5 0 == 0
>	больше чем	8 > 5 10 > 0
<=	меньше чем или равно	4 <= 5 8 <= 8
>=	больше чем или равно	9 >= 5 2 >= 2
!=	не равно	1 != 0 4 != 3.99

В программе GENIE1 из предыдущего раздела используется следующий условный оператор:

```
if(number<5)                // если (цифра <5)
{
    // Это число меньше, чем 5!
    printf("That number is less than 5!\n");
}
```

Первая строка начинается с ключевого слова `if`. После него идет неравенство в круглых скобках. Сравняется значение числовой переменной `number` с постоянным значением 5. Операция сравнения — “меньше чем”. Значение числовой переменной `number` — меньше чем 5? Если так, выполняется инструкция в фигурных скобках. В противном случае она опускается.

Рассмотрим следующую модификацию:

```
if(number==5)              // если (цифра ==5)
{
    // Это число равно 5!
    printf("That number is 5!\n");
}
```

Теперь неравенство выглядит так: `number==5`. (Введенное число равно пяти?) Если это так, инструкция `printf()` отображает `That number is 5!` (Это число равно 5!).

Следующий фрагмент программы (очередное изменение предыдущей) сравнивает значение числовой переменной `number` с 5:

```

if(number>=5)                                // если (цифра >=5)
{
    // Это число больше, чем 4!
    printf("That number is more than 4!\n");
}

```

На сей раз в качестве операции сравнения выступает “больше чем или равно”: введенное число равно 5 или больше, чем 5? Если значение числовой переменной `number` больше или равно 5, оно должно быть больше, чем 4, и инструкция `printf()` отображает эту важную информацию на экране.

В следующей модификации программы `GENIE1.C`, в отличие от предыдущих примеров, неравенство не изменяется. Зато в ней демонстрируется, что ключевому слову `if` может принадлежать не одна инструкция, а несколько:

```

if(number<5)                                // если (цифра <5)
{
    // Это число меньше, чем 5!
    printf("That number is less than 5!\n");
    // Клянусь, я само совершенство!
    printf("By goodness, aren't I smart?\n");
}

```

Все, что написано между фигурными скобками, выполняется, если неравенство истинно. В более сложных программах на C между фигурными скобками может стоять уйма инструкций; и поскольку они стоят между фигурными скобками, принадлежащими ключевому слову `if`, все они выполняются только в том случае, если неравенство истинно. (Именно поэтому они набраны с отступом — чтобы было видно, что все они принадлежат условному оператору.)

- ✓ Обычно в неравенстве сравнивается переменная со значением. Это может быть числовая или односимвольная переменная.
- ✓ Ключевое слово `if` не может сравнить строки. О сравнении строк рассказывается в моей книге *C All-in-One Desk Reference For Dummies*, выпущенной издательством “Wiley”.
- ✓ Со знаками `<` (меньше чем) и `>` (больше чем) и всей их родней вы должны быть знакомы по школьному курсу математики. В противном случае вы должны знать, что эти символы обозначают: символ `>` — больше чем, потому что сначала идет большая сторона; а символ `<` — меньше чем, потому что сначала идет меньшая сторона.
- ✓ Символы для “меньше или равно” и “больше или равно” всегда обозначаются так: `<=` и `>=`. Если порядок знаков перепутаете, компилятор сгенерирует сообщение об ошибке.
- ✓ В C “не” обозначается символом восклицательного знака. Так, `!=` означает “не равно”. А то, что `!TRUE` (**ИСТИНА**) (не является истинным высказыванием), является `FALSE` (**ЛОЖЬ**). “Возможно, вы думаете, что это — масло, но это **!** масло”. Нет, я думаю, я **!** хочу есть сырые чипсы.
- ✓ Чтобы сравнить два значения на равенство, используйте два знака “`=`”. Чтобы написать условный оператор, который проверяет, равны ли две вещи, ставьте два знака “`=`”. Например, если хотите проверить, равно ли `x` числу 5, напишите:

```

if (x==5)                                    // если (x == 5)

```

Читать эту инструкцию можно так: “Если значение переменной `x` равно 5, то...” или “Если значение переменной `x` равняется 5, то...” Но ни в коем случае не оставляйте в неравенстве только один знак “`=`” — это ужасная ошибка.





- ✓ Если вы используете только один знак "=", а не два, сообщения об ошибке вы не получите, однако программа будет неправильна (во врезке "Операции сравнения: не забивайте себе голову этой ерундой" объясняется, почему).
- ✓ Если вы ранее программировали на других машинных языках, имейте в виду, что в языке C слова `2ewd` или `fi` не предусмотрены. Конец условного оператора компилятору указывает закрывающая фигурная скобка.
- ✓ Кроме того, после ключевого слова `if` не используется ключевое слово `then`, как это имеет место в инструкции `if-then` в языках программирования Pascal (Паскаль) или BASIC (БЕЙСИК).



Операции сравнения: не забивайте себе голову этой ерундой

То, что мы называли неравенством в условном операторе, может не содержать никаких символов вообще! Странно, но истинно. Компилятор с языка C лишь вычисляет то, что находится между круглыми скобками. Затем он выясняет, является ли это выражение истинным или ложным.

Если неравенство содержит знак `<`, `>`, `==` или любой другой из табл. 12.1, то компилятор выясняет, является неравенство истинным или ложным. Однако между круглыми скобками может стоять любая правильная (допустимая) инструкция языка C; компилятор просто определит, к чему приводится вырабатываемое ею значение — к истине или ко лжи. Например:

```
if(input=1) // если (input=1)
```

Этот условный оператор не выясняет, равно ли 1 значение переменной `input` (ввод). Нет, в данном случае нужно было бы написать два знака `==`. Вместо этого вот что происходит между круглыми скобками: числовой переменной `input` присваивается значение 1. Это то же самое, что

```
input=1;
```

Компилятор с языка C повинуется этой команде, запоминая 1 в переменной `input`. Затем он размышляет: "Это истина или ложь?" и полагает, что инструкция должна быть истиной. Именно это он и говорит ключевому слову `if`, а затем выполняются все инструкции, которые принадлежат условному оператору.

Форматирование условного оператора



Условный оператор — ваша первая "сложная" инструкция языка C. В языке C таких инструкций много, но `if` — первая и, возможно, самая популярная, хотя я сомневаюсь, что для слов языка программирования было когда-либо проведено сравнение по популярности.

Хотя пока вы, вероятно, видели условный оператор только с фигурными скобками, его можно также записать как обычную инструкцию языка C. Например, рассмотрим следующую модификацию фрагмента программы GENIE1:

```
if(number==5) // если (цифра ==5)
{
    // Это число равно 5!
    printf("That number is 5!\n");
}
```

В С совершенно законно записать этот оператор так, как записываются обычные инструкции. Вот так:

```
// Это число равно 5!  
if(number==5) printf("That number is 5!\n");
```

Эта строка больше походит на инструкцию языка С. Она заканчивается точкой с запятой. Все работает так, как и прежде: если значение переменной `number` (число) равно 5, выполняется инструкция `printf()`. Если значение переменной `number` не равняется 5, остальная часть инструкции пропускается.

Хотя все это допускается при программировании на С, я рекомендую использовать фигурные скобки в ваших условных операторах, во всяком случае до тех пор, пока вы не научитесь бегло читать программы на языке С.

Окончательное решение проблемы подоходного налога

Я изобрел, на мой взгляд, самый справедливый и, очевидно, наиболее полный благих намерений способ решить, кто должен платить наибольшие подоходные налоги. Вы должны платить тем больше налогов, чем вы выше, и тем больше налогов, чем теплее на улице. Да, было бы трудно уклониться от такого законодательства.

Для решения этой проблемы идеально подходит ключевое слово `if`. Вы платите налоги в зависимости от роста и температуры на улице, умноженной на ваше любимое число и затем на 10. Налог, который вы платите, равен большему из двух вычисленных таким способом чисел (одно вычисляется для роста, второе — для температуры). Чтобы выяснять, какое число больше, в программе `TAXES.C` используется ключевое слово `if` с символом “больше чем”. Операция сравнения выполняется дважды — один раз для значения роста, а второй раз — для температуры на улице:

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main() // главная функция  
{  
    int tax1,tax2;  
    char height[4],temp[4],favnum[5];  
  
    // Введите ваш рост в дюймах  
    printf("Enter your height in inches:");  
    gets(height); // рост  
    // Какая температура на улице?  
    printf("What temperature is it outside?");  
    gets(temp); // временная переменная  
    // Введите ваше любимое число  
    printf("Enter your favorite number:");  
    gets(favnum);  
  
    tax1 = atoi(height) * atoi(favnum);  
    tax2 = atoi(temp) * atoi(favnum);  
  
    if(tax1>tax2) // если (tax1> tax2)  
    {  
        // Вы должны уплатить $%d налога  
        printf("You owe $%d in taxes.\n",tax1*10);  
    }  
    if(tax2>=tax1) // если (tax2>=tax1)  
    {
```



```
// Вы должны уплатить $%d налога
    printf("You owe $%d in taxes.\n", tax2*10);
}
return(0);
}
```

Эта программа — одна из наиболее длинных в этой книге. Будьте очень внимательны при вводе. В ней нет ничего нового, но она охватывает почти все, что было изучено в нескольких начальных главах. Перепроверьте каждую строку, когда будете вводить эту программу с помощью вашего редактора.

Сохраните файл на диске под именем TAXES.C.

Скомпилируйте TAXES.C. Исправьте все обнаруженные ошибки.

Выполните программу. У вас получится:

Enter your height in inches:

(Введите ваш рост в дюймах.)

Введите ваш рост в дюймах. Пять футов — 60 дюймов; шесть футов — 72 дюйма. Средний рост человека — 67 дюймов. Нажмите <Enter>.

What temperature is it outside?

(Какая температура на улице?)

Прямо сейчас, среди зимы на Тихоокеанском Северо-Западе 18 градусов. Это — по Фаренгейту, между прочим. Не смейте вводить меньшее число по Цельсию. Если вы это сделаете, налоговая служба выследит вас подобно провинившейся музыкальной звезде и заставит вас платить, платить, платить.

Enter your favorite number:

(Введите ваше любимое число:)

Введите ваше любимое число. Мое — 11. Нажмите <Enter>.

Если я введу 72 (мой рост), 18 и 11, например, я увижу следующий результат — сумму, которую мне придется уплатить 15 апреля:

You owe \$7920 in taxes.

(Вы должны уплатить 7920\$ налога.)

Ого! А я-то думал, что старая система была плоха. Кажется, мое любимое число уменьшилось.



- ✓ Во втором неравенстве используется знак “больше чем или равно”. Оно учитывает случай, когда рост равен температуре. Если значения равны, то значения переменных `tax1` и `tax2` также равны. Первое неравенство “`tax1` больше, чем `tax2`” ложно, потому что переменные `tax1` и `tax2` также равны. Второе неравенство “`tax1` больше или равно `tax2`” выполнено, когда `tax1` больше, чем `tax2`, или когда эти значения равны.
- ✓ Если вы введете нуль в качестве вашего любимого числа, программа говорит, что вы ничего не должны налоговой службе. К сожалению, налоговая система не позволяет вам любить нуль или какие-либо отрицательные числа. Грустно, но истинно.

Что делать, если неравенство не выполняется?

Продолжаем решать налоговую проблему!

Нет, не ту, которую создало правительство. Мы изучаем исходный текст TAXES.C, представленный в предыдущем разделе. Откройте его в вашем текстовом редакторе для редактирования.

Последняя часть программы TAXES.C состоит из двух условных операторов. Второй условный оператор на самом деле не нужен. Вместо него можно использовать в программе другое слово языка C — `else`.

Давайте в программе TAXES.C заменим строку

```
if (tax2 >= tax1) // если (tax2 >= tax1)
```

Отредактируйте эту строку: удалите ключевое слово `if` (если) и неравенство в круглых скобках и замените их вот чем:

```
else // иначе
```

Вот именно — только одним словом `else`. Никаких неравенств и никаких точек с запятыми, всего лишь проверьте, что вы напечатали это слово на нижнем регистре.

Сохраните файл на диске.

Скомпилируйте TAXES.C. Выполните машинную программу. Вывод будет тот же самый, потому что программа не изменилась (если, конечно, на улице не стало теплее или вы не выросли). Просто вы создали *конструкцию if-else*, которая представляет собой еще один способ записи процесса принятия решений в программах на C.

- ✓ Ключевое слово `else` и его инструкции — вторая, дополнительная часть инструкции `if`. В ней указываются те инструкции, которые должны быть выполнены, если условие не выполнено.
- ✓ `else` — иначе.

Охват всех возможностей с помощью ключевого слова `else`

Комбинация ключевых слов `if-else` позволяет записать программу, которая может принимать альтернативные решения. Само по себе ключевое слово `if` может помочь принять решение и выполнить специальные команды, если условие выполнено. Но если в условном операторе есть еще и фраза `else`, программа выберет один из двух наборов операторов в зависимости от результата операции сравнения, записанной в неравенстве. На рис. 12.2 показано, как это все работает.

Если неравенство истинно, выполняются те инструкции условного оператора, которые принадлежат конструкции `if`. Но если неравенство является ложным, выполняются инструкции, принадлежащие конструкции `else`. Выполнение программы идет по одной из ветвей, как показано на рис. 12.2. Затем после выполнения конструкции `if-else` выполняется инструкция после заключительной фигурной скобки, относящейся к конструкции `else`.

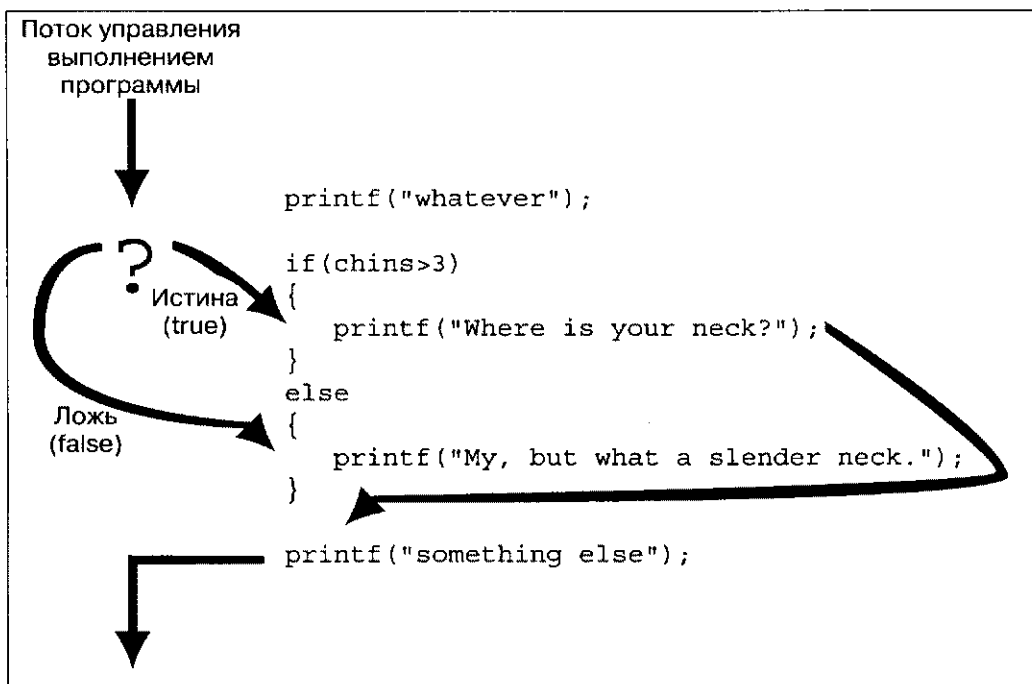


Рис. 12.2. Выполнение конструкции if-else

Формат условного оператора с конструкцией else

Ключевое слово `else` может использоваться в условном операторе. Ключевое слово `else` имеет свою собственную группу инструкций, которые выполняются только в том случае, если неравенство не выполняется. Вот — формат:

```

if(неравенство)                // если (неравенство)
{
    инструкция (и);
}
else                            // иначе
{
    инструкция (и);
}

```

Ключевое слово `if` проверяет неравенство в круглых скобках. Если это неравенство выполнено, выполняются инструкции, которые следуют в фигурных скобках сразу после ключевого слова `if` данного условного оператора. Но если неравенство является ложным, выполняются те инструкции, которые заключены в фигурные скобки, следующие после ключевого слова `else`. Так или иначе, выполняется только одна группа инструкций, а не обе.

Ключевое слово `else`, как и все другие слова на языке C, вводится на нижнем регистре. После него не ставится точка с запятой. Вместо точки с запятой за ним следует пара фигурных скобок. В них (в фигурных скобках) заключена одна или несколько инструкций, которые будут выполнены только в том случае, если неравенство после ключевого слова `if` не выполнено. Обратите внимание, что каждая из этих инструкций должна заканчиваться точкой с запятой, повинаясь законам C, выгравированным на древних камнях при закладке основ языка еще за три десятилетия до конца прошлого тысячелетия.

Инструкции, принадлежащие ключевому слову `else`, выполняются только в том случае, если условие, проверяемое ключевым словом `if`, является ложным. В табл. 12.2 приведены отрицания условий, которые определяются операциями сравнений, проверяемых ключевым словом `if`.

Таблица 12.2. Операции сравнения, проверяемые ключевым словом `if`, и их отрицания

Знак неравенства, проверяемого ключевым словом <code>if</code>	Инструкция <code>else</code> выполняется при этом условии
<	>= (больше или равно)
==	!= (не равно)
>	<= (меньше или равно)
<=	> (больше чем)
>=	< (меньше чем)
!=	== (равно)



- ✓ Ключевое слово `else` используется только с ключевым словом `if`.
- ✓ В фигурные скобки, принадлежащие ключевым словам `if` и `else`, можно заключить не только одну, но и несколько инструкций. Инструкции, принадлежащие ключевому слову `if`, выполняются только тогда, когда неравенство истинно; инструкции же, принадлежащие ключевому слову `else`, выполняются только тогда, когда неравенство является ложным.
- ✓ *Выполнить* означает заставить работать. Программы на С выполняют, т.е. заставляют работать инструкции одна за другой, начиная от начала исходного текста (первая строка). Каждая строка выполняется одна за другой, пока не встретится инструкция, подобная `if` или `else`. В этом случае программа выполняет ту или иную инструкцию в зависимости от результата операции сравнения в инструкции `if`.
- ✓ Если в программе не требуется принятие альтернативного решения, конструкция `else` не нужна. Например, в программе TAXES конструкция `else` нужна именно потому, что в данной программе необходимо принять альтернативное решение. Однако предположим, что вы пишете программу, которая должна продолжить свое обычное выполнение, даже если условие не выполняется. В этом случае конструкция `else` не нужна, если ошибка не происходит, поскольку в этом случае программа.



- ✓ Если вы знакомы с другими языками программирования, обратите внимание, что в языке С в конце условного оператора никакого ключевого слова вроде `end-else` не нужно. И вообще, С — это не отвратительная версия старого Паскаля, к вашему сведению. В конце инструкции `else` стоит заключительная фигурная скобка, она и заканчивает эту инструкцию — все происходит в полной аналогии с инструкцией `if`.



Форматирование: эти прекрасные безделушки

Конструкция `if-else` не всегда записывается с изящными фигурными скобками. Точно так же, как можно сократить условный оператор, записав его в одной строке, можно сократить и инструкции `else`. Делать это я не рекомендую. Вот главная часть программы `TAXES.C`:

```
if(tax1>tax2)                                // если (tax1> tax2)
{
    // Вы должны уплатить $%d налога
    printf("You owe $%d in taxes.\n",tax1*10);
}
else                                          // иначе
{
    // Вы должны уплатить $%d налога
    printf("You owe $%d in taxes.\n",tax2*10);
}
```

Здесь вы видите суть программы `TAXES.C`: конструкцию `if-else`. Поскольку и инструкции `if`, и инструкции `else` принадлежит только по одной инструкции, данный фрагмент исходного текста можно сократить так:

```
if(tax1>tax2)                                // если (tax1> tax2)
// Вы должны уплатить $%d налога
printf("You owe $%d in taxes.\n",tax1*10);
else                                          // иначе
// Вы должны уплатить $%d налога
printf("You owe $%d in taxes.\n",tax2*10);
```

В этом формате, как и в приведенном выше, сохраняются отступы, благодаря которым становится ясно, что чему принадлежит (и, кроме того, легко распознать конструкцию `if-else`). Допустим также следующий формат, хотя он и затрудняет чтение программы:

```
if(tax1>tax2) printf("You owe $%d in taxes.\n",tax1*10);
else printf("You owe $%d in taxes.\n",tax2*10);
```

Все буквально раздавлено на двух строках; условный оператор размещен на своей собственной строке, а инструкции `else` — на своей. Обе строки заканчиваются точкой с запятой, благодаря чему может создаться впечатление, что работают две инструкции на языке `C`. Однако это слишком длинно! Пожалуйста, не пишите свои программы подобным образом!

Вы можете использовать эту уловку — опустить фигурные скобки, если только одна инструкция принадлежит ключевому слову `if` или ключевому слову `else`. Если же нужно выполнить несколько инструкций, по законодательству языка `C` фигурные скобки опустить нельзя. Именно поэтому я рекомендую пользоваться ими все время, не стоит подвергать себя риску поремного заключения из-за краткости. Вот пример правильной, но несколько рискованной конструкции:

```
if(tax1>tax2)                                // если (tax1> tax2)
// Вы должны уплатить $%d налога
printf("You owe $%d in taxes.\n",tax1*10);
else                                          // иначе
{
    // Вы должны уплатить $%d налога
    printf("You owe $%d in taxes.\n",tax2*10);
    // Платить за жизнь в таком холоде?!
    printf("It pays to live where it's cold!\n");
}
```

Поскольку две инструкции `printf` принадлежат предыдущему ключевому слову `else`, фигурные скобки обязательны.

Странный случай комбинации else-if и множественные альтернативные решения

Язык C богат конструкциями, предназначенными для принятия решений. Ключевое слово `if` помогает проверить только одно условие. Ключевое слово `if` помогает принять решение в зависимости от того, истинно или ложно некоторое условие. И, если проверяемое условие истинно, выполняется некоторая группа инструкций. Иначе эта группа инструкций пропускается. (После выполнения группы инструкций, принадлежащей ключевому слову `if`, выполнение программы продолжается как обычно.) Альтернативные условия — вот для чего предназначен дуэт `if-else`. Так или иначе, но выполняется только один набор инструкций, а не другой: какой именно — зависит от результата операции сравнения.

А что если нужно принять решение иного типа: “выполнить первый, второй или третий набор инструкций”? Для принятия решений такого типа нужна удивительная и чрезвычайно универсальная комбинация `else-if`. Такая комбинация действительно может свести с ума, но она удобна.

Следующая программа — модификация исходного текста `GENIE1.C`, приведенного ранее в этой главе. На сей раз используется комбинация `else-if`. Благодаря этой комбинации компьютерный джин может точно сообщить: число меньше 5, равно 5 или больше 5.

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

int main()                                // главная функция
{
    char num[2];
    int number;

    // Я - твой компьютерный джин!
    printf("I am your computer genie!\n");
    // Введите число от 0 до 9:
    printf("Enter a number from 0 to 9:");
    gets(num);                             // цифра
    number=atoi(num);
    if(number<5)                           // если (число <5)
    {
        // Число меньше 5!
        printf("That number is less than 5!\n");
    }
    else if(number==5)                     // иначе если (число == 5)
    {
        printf("You typed in 5!\n");       // Вы ввели 5!
    }
    else                                   // иначе
    {
        // Число больше 5!
        printf("That number is more than 5!\n");
    }

    // Джин знает все, видит все!
    printf("The genie knows all, sees all!\n");
    return(0);
}
```

Начните работу над этим исходным текстом: загрузите исходный текст `GENIE1.C`, созданный ранее в ходе изучения этой главы. Сделайте такие изменения, чтобы получился новый исходный текст `GENIE2.C`, приведенный выше.

Соблюдайте выравнивание. Обращайте внимание на все; в неравенстве, принадлежащем комбинации `else-if`, должны быть два знака `"=`". Обратите внимание, где нужны точки с запятой и где они не нужны.

После вставки новых строк сохраните файл на диске под именем `GENIE2.C`.

Скомпилируйте `GENIE2.C`. Если уродливые ошибки-монстры поднимут свои головы, заново отредактируйте исходный текст и затем перетранслируйте его.

Выполните полученную программу и убедитесь сами, насколько улучшилось зрение у этого слепого компьютерного ясновидца. Напечатайте 3 и убедитесь, что джин скажет: `That number is less than 5!` (Это число меньше 5!). Напечатайте 9 и убедитесь, что джин скажет: `That number is more than 5!` (Это число больше 5!). Напечатайте 5 и убедитесь, что джин знает: `You typed in 5!` (Вы напечатали 5!).

- ✓ Неравенство в комбинации `else-if` напоминает объединение ключевого слова `else` и условного оператора. Второе ключевое слово `if` следует сразу после пробела, который стоит после ключевого слова `else`. Затем следует его собственная инструкция сравнения, в результате вычисления которой получается истина (`true`) либо ложь (`false`).
- ✓ В `GENIE2.C` неравенство в комбинации `else-if` (`number==5`) проверяет, равно ли значение переменной `number` числу 5. Операция сравнения обозначена двумя знаками `"="`.
- ✓ Если хотите, вы можете целыми днями писать комбинации `else-if`, `else-if`, `else-if`, однако язык `C` имеет лучшее решение — инструкцию выбора `select-case`.

Еще одна программа: подумать только, и в самом деле хитрейший джин

Решение существует всегда². Если вы хотите, вы можете написать программу, которая с помощью ключевого слова `if` сравнивает любое значение, от нуля до бесконечности, или от бесконечности до нуля, и “компьютерный джин” точно угадал бы введенное число. Но к чему здесь ключевое слово `if` вообще?

Ключевое слово `if` — тема этого раздела, наряду с инструкцией `if-else` и комбинацией `else-if` и так далее. Но в следующем исходном тексте `GENIE3.C` ключевое слово `if` не используется вообще. Эта программа обманывает так, что джин всегда угадывает точно:

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

```
int main()                                // главная функция
(
    char num;

    // Я - твой компьютерный джин!
    printf("I am your computer genie!\n");

    // Введите число от 0 до 9:
    printf("Enter a number from 0 to 9:");
    num = getchar();                       // цифра

    // ("Вы напечатали %c!\n", цифра)
    printf("You typed in %c!\n", num);
```

² Да, автор не лишен чувства юмора и в этом вопросе! Читайте дальше, и убедитесь в этом сами. — Прим. ред.

```
// Джин знает все, видит все!
printf("The genie knows all, sees all!\n");
return(0);
}
```

Вы можете создать этот исходный текст, редактируя GENIE1.C или GENIE2.C. Сделайте необходимые изменения, а затем сохраните исходный текст на диске под именем GENIE3.C.

Скомпилируйте программу. Исправьте все ошибки, хотя я надеюсь, что хоть в этот раз их не будет. Выполните полученную программу:

```
I am your computer genie!
Enter a number from 0 to 9: 8
You typed in 8!
The genie knows all, sees all!
```

(Я — твой компьютерный джин!

Введите число от 0 до 9: 8

Вы напечатали 8!

Джин знает все, видит все!)

Выполните программу снова и снова, вводя различные числа. Эй! Этот джин знает точно, что вы напечатали! Интересно, как это случилось? Это произошло, несомненно, вследствие ловкого использования языка C. Вы приручили компьютер!

- ✓ В чем проблема с GENIE3.C? Эта программа не принимает решение. Джин не совсем не хитер — он только повторяет то, что вы уже знаете. А вот ценность ключевых слов `if`, `else` и других именно в том и состоит, что они позволяют принимать решение в программе.
- ✓ Функция `getchar()` более подробно описана в главе 10 "Переменные типа `char`".

Сравнение символов с помощью ключевого слова `if`

В этой главе...

- Сравнение символов с помощью ключевого слова `if`
- Стандартный ввод
- Обход недостатков `getchar()`
- Принятие решений типа “да или нет”

Оставим извинения для предыдущей главы. Да, я определенно злоупотребляю числами при представлении достоинств команды `if`. Не все компьютерные программы сконцентрированы на числах, и применение ключевого слова `if` не ограничивается сравнением сумм долларов, эклиптических орбит, масс субатомных частиц или калорий различных сортов печенья. Нет, с помощью условного оператора сравнивать можно также символы алфавита. Это должно, наконец, открыть таинственную причину того, что Т “больше чем” S и почему знак доллара меньше, чем знак “минус”. В этой главе выясняются подробности.

Мир ключевого слова `if` без числовых значений

Позвольте спросить: как можно сравнивать два символа алфавита, например, две буквы? Действительно, эта тема немного связана с вопросами: “Сколько ангелов может танцевать на острие иголки?” и “Что это за танец — какой-нибудь традиционный, гавот, рок или, возможно, небесное хоки-поки?”. Все же необходимо решить задачу сравнения символов. Если вы записываете в программе меню, вы должны знать, что выбирает пользователь — опцию А, В или С. Это удобно сделать с помощью ключевого слова `if`. Например, вот так:

```
if (key == 'A')           // если (клавиша == 'A')
```

Здесь `key` (клавиша) — односимвольная переменная, хранящая символ, введенный с клавиатуры. Сравнение, выполняемое с помощью ключевого слова `if`, позволяет определить, равно ли (два знака “=”) содержимое этой переменной символу А, который заключен в одинарные кавычки. Эта операция сравнения законна. Пользователь напечатал А?

- ✓ Чтобы сравнить односимвольную переменную с символом — буквой, цифрой или другим символом, — сравниваемый символ нужно заключить в одинарные кавычки. Запомните: один (отдельный) символ — одинарные кавычки.
- ✓ Чтобы с помощью ключевого слова `if` проверить, равны ли два выражения, используются два знака “=”. Не забывайте, что для обозначения операции сравнения используются два знака “=”, а не один.



- ✓ Когда сравниваете переменную (числовую или односимвольную) с константой, в операции сравнения сначала всегда указывается переменная.
- ✓ В хороший солнечный день 4,9Е3 ангелов может танцевать на острие иглолки — все зависит от относительных размеров иглолки и размеров ангелов, а также от характера танца — танцуют ли они все вместе или по очереди.
- ✓ Сравнивая два символа или числа, вы в действительности сравниваете их значения в АSCII-коде. Это один из тех необычных случаев, когда односимвольная переменная ведет себя так, что она более похожа на целое число, чем на букву, цифру или символ.



Компьютеры и математика (неужели я все еще должен напоминать, что вы можете пропустить этот материал?)

С грустью должен сказать, что слишком многое в компьютерах действительно связано с математикой. Предшественником компьютера был калькулятор, идеи которого были заложены еще в абак, который, так или иначе, связан с человеческими пальцами и бегунками. В конце концов, обработка чисел называется¹ словом *compu-ting* (вычисление), которое происходит от древнего латинского термина *computare*. Это слово буквально означает "наймем еще несколько бухгалтеров, и мы никогда не будем знать, что происходит в действительности". А, кроме того, вычисления связаны с цифрами. Так что компьютеры связаны с цифрами.

Что больше: S или T, \$ или —?

Действительно ли T больше, чем S? В алфавитном порядке, да. Ведь T следует после S. Но что можно сказать относительно знака доллара и знака "минус"? Какой из них больше? И почему Q должно быть меньше, чем U, — только потому, что каждый знает, что U всегда следует за Q? Очень сомнительно.

Чтобы разгадать эту большую тайну жизни, я представлю вам исходный текст программы GREATER.C. Она просит ввести два отдельных символа. Эти символы сравниваются с помощью условного оператора — конструкции if-else. Затем отображается больший из этих двух символов. Хотя эта программа не решает проблему "ангелов на острие иглолки", она помогает разрешить некоторые алфавитные загадки:

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

```
int main() // главная функция
{
    char a,b; // символ a, b

    // Какой символ больше?
    printf("Which character is greater?\n");
    printf("Type a single character:"); // Введите один символ:
    a=getchar();
    printf("Type another character:"); // Введите еще один символ:
    b=getchar();

    if(a > b) // если (a > b)
    {
        // (" '%c' больше чем '%c'!\n", a, b)
        printf("' %c' is greater than '%c'!\n",a,b);
    }
}
```

¹ В английском языке. — Прим. ред.

```

else if (b > a)                // иначе если (b>a)
{
// (" '%c' больше чем '%c'!\n ", b, a)
    printf("'%c' is greater than '%c'!\n",b,a);
}
else                            // иначе
{
// В следующий раз не печатайте тот же самый символ дважды.
    printf("Next time, don't type the same character twice.");
}
return(0);
}

```

Введите исходный текст GREATER.C с помощью вашего редактора. Удостоверьтесь, что вы ввели все необходимые фигурные скобки, что все точки с запятой находятся на своих местах, что двойные кавычки расставлены правильно, а также обратите внимание на другие незначительные нюансы грамматики (источники возможных неприятностей) языка C.

Сохраните файл на диске под именем GREATER.C.

Скомпилируйте GREATER.C.

Выполните полученную программу. Она задаст вопрос:

Which character is greater?

Type a single character:

(Какой символ больше?

Напечатайте один символ:)

Напечатайте символ, например \$ (знак доллара). Нажмите <Enter> после ввода символа.

Мм-да! Что случилось? Вы увидите примерно следующее:

Type another character: '\$' is greater than '

'\$

What? What? What?

(Напечатайте еще один символ: '\$', больше чем '

'\$

Чепуха? Чепуха? Чепуха?)

Программирование как род деятельности почти непредсказуемо! Проблема. Прямо посередине главы, в которой говорится о сравнении односимвольных переменных, я должен полностью прервать разговор по теме и рассказать кое-что важное о программировании на C: о правильном способе чтения отдельных символов с клавиатуры.

Проблема с getchar()

Вопреки тому, что говорится в некоторых брошюрах, `getchar()` не читает один отдельный символ с клавиатуры. Отдельный символ — это то, что возвращает функция `getchar()`, но это совсем не то, что делает эта функция.

На самом деле `getchar()` читает то, что называется *стандартным вводом*. Данная функция захватывает первый напечатанный символ и сохраняет его, но после этого переходит в состояние ожидания и ждет, когда вы нажмете клавишу, которая играет роль сигнала “конец”. В качестве конца ввода воспринимаются два символа. Один из них вводится нажатием клавиши <Enter>. Второй символ — это символ конца файла, EOF (end-of-file — конец файла). В Windows это символ <Ctrl+Z>. В операционных системах семейства Unix это символ <Ctrl+D>.

Выполните программу GREATER из предыдущего раздела. Когда она попросит ввести символ, напечатайте 123 и затем нажмите клавишу <Enter>. Вот что вы увидите:

Which character is greater?

Type a single character:123

Type another character:'2' is greater than '1'

(Какой символ больше?)

Напечатайте один символ:123

Напечатайте еще один символ:'2' больше, чем '1')

Когда сначала нужно было напечатать один символ, вы в стандартный ввод для программы ввели строку 123 плюс символ, который вводится нажатием клавиши <Enter>.

Функция `getchar()` читает стандартный ввод. Сначала она прочитает 1 и поместит этот символ в переменную `a`. (Строка `a=getchar()`; в GREATER.C). Затем программа использует второй вызов функции `getchar()`, чтобы снова прочитать символ со стандартного ввода — но он уже там есть; поэтому 2 читается в переменную `b` (строка `b=getchar()`).

А куда девалось 3? Оно игнорируется; ведь нет места, чтобы сохранить его. Но нажатие клавиши <Enter> указывает программе, что символы в стандартный ввод были введены, и потому программа продолжает выполнение, немедленно отображая результат, причем все отображается на одной — последней — строке:

Type another character:'2' is greater than '1'!

(Напечатайте еще один символ:'2' больше, чем '1!')

Вы можете подумать, что обойти эту проблему можно следующим образом: чтобы закончить стандартный ввод, нужно нажать клавишу <Enter> сразу после ввода первого символа. В конце концов, ведь именно клавиша <Enter> закрывает стандартный ввод. Но этот метод не работает.

Выполните снова программу GREATER; на сей раз введите \$ и нажмите клавишу <Enter>, когда появится первая подсказка. Вот каким будет вывод:

Which character is greater?

Type a single character:\$

Type another character:'\$' is greater than '1'!

(Какой символ больше?)

Напечатайте один символ:\$

Напечатайте еще один символ:'\$' больше, чем '1'!

Это примерно соответствует тому, что вы получили в первый раз, когда ввели программу. Видите, что отображается в результате нажатия клавиши <Enter>? Это — то, что отделяет ' в конце одной строки от ' в начале следующей строки.

В этом случае символ, соответствующий нажатию клавиши <Enter>, принимается за символ в стандартном вводе. В конце концов, клавиша <Enter> — обычная клавиша на клавиатуре и ей соответствует код символа, точно так же как и любой другой клавише. Поэтому клавишу <Enter> нельзя использовать для надлежащего способа завершения стандартного ввода. Она просто помещает символ конца файла (EOF).

Выполните программу GREATER еще раз:

Which character is greater?

Type a single character:

(Какой символ больше?)

Напечатайте один символ:

В Windows напечатайте \$ и затем нажмите <Ctrl+Z> и <Enter>.

В Unix напечатайте \$ и затем нажмите <Ctrl+D>.

Type another character:

(Напечатайте еще один символ:)

Теперь можете напечатать 2 и нажать <Enter>:

```
'2' is greater than '$'!
```

```
('2' больше, чем '$'!)
```

Наконец, программа выполняется должным образом, но только после того, как вы решили “в лоб” ввести конец стандартного ввода, чтобы `getchar()` могла должным образом обработать его. Очевидно, если бы пользователям ваших программ пришлось объяснять нечто подобное, они сочли бы ваши программы, по меньшей мере, надоедливыми. Есть лучшее решение. Продолжайте читать.

- ✓ В Windows символ конца файла EOF (end-of-file, конец файла) — <Ctrl+Z>. Это — код 27 в таблице ASCII.
- ✓ В Unix (Linux, FreeBSD, Mac OS и т.д.) символ конца файла EOF (end-of-file, конец файла) — <Ctrl+D>. Это — код 4 в таблице ASCII.
- ✓ Чтобы должным образом ввести <Ctrl+Z> в Windows, вы обязаны нажать <Ctrl+Z>, а затем клавишу <Enter>. В Unix комбинация <Ctrl+D> сработает немедленно.
- ✓ Будьте внимательны, когда нажимаете <Ctrl+D> в приглашении к вводу команды Unix! Во многих случаях клавиша <Ctrl+D> также заканчивает работу среды, отключая вас от системы.

Модификация GREATER.C: правильное считывание стандартного ввода

Проще всего исправить GREATER.C следующим методом: необходимо очистить (сбросить состояние) весь стандартный ввод перед тем, как второй вызов функции `getchar()` приступит к его чтению. Благодаря этой очистке старые символы, сохранившиеся в нем по ошибке, не будут прочитаны.

Функция языка C `fflush()` как раз и убирает информацию, которая ожидает записи в файл. Поскольку C обрабатывает стандартный ввод как файл определенного типа, для очистки стандартного ввода можно применить `fflush()`. Вот формат вызова:

```
fflush(stdin);
```

Эту инструкцию можно вставить в любое место любой программы, чтобы гарантировать, что никакие неиспользованные оставшиеся символы не будут случайно считаны с клавиатуры. Таким образом, это очень полезная инструкция.

Чтобы исправить исходный текст GREATER.C, нужно вставить функцию `fflush()` после первого вызова функции `getchar()`. Ниже показан исправленный фрагмент исходного текста:

```
printf("Type a single character:");          // Введите один символ:
a=getchar();
fflush(stdin);
printf("Type another character:");          // Введите еще один символ:
b=getchar();
```

Фактически к исходному тексту GREATER.C пришлось добавить новую строку `fflush(stdin);`. Сохраните измененный файл на диске. Скомпилируйте и выполните полученную программу. Программа теперь выполняется должным образом, а вы можете продолжить читать о сравнении символов с помощью команды `if`.

Если функция `fflush(stdin)` не поможет, замените ее следующей функцией:

```
fpurge(stdin);
```



Функция `fpurge()` уж определенно сотрет текст, ждущий считывания. Я заметил, что именно эта функция требуется для программ, выполняемых в среде Unix, Linux и Mac OS.

Чтение одиночных символов: функции `getchar()`, `getch()` и `getche()`

Можно ли с помощью `getchar()` прочитать только один символ? Увы, функция `getchar()` не предназначена для чтения символов с клавиатуры. В действительности она читает стандартный ввод, который почти во всех компьютерах представляет собой текст, вводимый с клавиатуры.

Чтобы читать с клавиатуры отдельные символы, определены иные функции. В некоторых версиях GCC для Windows имеются функции `getch()` и `getche()`, которые могут читать текст непосредственно с клавиатуры. Эти функции свободны от недостатка, связанного с чтением стандартного ввода с помощью функции `getchar()`. Но проиллюстрировать применение этих функций не так-то просто. Проблема состоит в том, что они не имеют соответствий в Unix.

Чтобы выполнить непосредственное чтение символов с клавиатуры, в Unix необходимо получить доступ к используемому терминалу, а затем интерпретировать генерируемые клавиатурой коды. Другое решение состоит в том, чтобы использовать библиотеку Curses.

БОЛЬШАЯ проблема в программе GREATER: два алфавита в ASCII — для строчных и прописных букв

Теперь, когда вы, возможно, справились с проблемой, связанной с `getchar()` в программе GREATER, пришло время исследовать вывод. Выполните программу снова. Попробуйте узнать, какой символ больше: '-' или '\$'.

```
Which character is greater?
```

```
Type a single character:-
```

```
Type another character:$
```

```
'-' is greater than '$'!
```

(Какой символ больше?

Напечатайте один символ:-

Напечатайте еще один символ:\$

'-' больше, чем '\$!')

И нам осталось выяснить, почему.

Видите ли, команда `if` ничего не знает о буквах, цифрах или символах. Вместо того чтобы сравнивать смысл символов, команда `if` сравнивает значения ASCII-кодов, соответствующих символам.

Значение кода ASCII для знака “минус” — 45. Значение кода для знака доллара — 36. Поскольку 36 меньше, чем 45, компьютер думает, что '-' больше, чем '\$'. Это справедливо также же для символов алфавита (букв) и их значений в коде ASCII.

В реальной жизни вы редко сравниваете один символ с другим. Вместо этого вы сравниваете нажатую клавишу с известным, допустимым выбором (эта тема рассматривается в следующем разделе).

✓ ASCII-коды приведены в приложении Б “Таблица ASCII”.

- ✓ Выполните программу снова и попробуйте ввести следующие два символа: **a** (строчная (маленькая) буква *a*) и **Z**. Прописная (большая) буква *Z* — меньше, чем строчная (маленькая) буква *a*, несмотря на то, что строчная (маленькая) буква *a* предшествует *Z* в алфавите. Причина заключается в том, что код ASCII имеет два алфавита: один для прописных букв и другой для нижнего регистра. Прописные буквы имеют меньшие значения, чем символы нижнего регистра, так что строчные буквы “a-z” всегда больше, чем прописные “A-Z”.



Природа алфавитного порядка — очень скучные мелочи

Итак, почему сначала идут А, В, С и почему Z стоит в алфавите последним? Ответ наглухо похоронен в груди мелочей, которые так любят запоминать большинство компьютерных маньяков. Поскольку я был любопытен, я подумал, что и мне нужно найти ответ на этот вопрос. И вот что я нашел.

Наш алфавит основан на древних алфавитах, которые в свою очередь основаны на еще более древних алфавитах, принесенных, возможно, пришельцами еще в эпоху динозавров. В ту древнюю эпоху символы алфавитов были символами, обозначающими различные вещи, которые люди использовали каждый день, и потому символы часто назывались так же, как и сами вещи: символ А обозначал вода и имел форму, подобную этому важному животному. Символ В обозначал дом и имел форму, подобную двери. И так далее для всех символов. Так это было для большинства ранних семитских языков, построенных по фонетическому принципу. (В фонетическом письме буквы — символы алфавита — обозначают звуки, а не являются пиктограммами или идеограммами.)

Греки заимствовали свой алфавит от семитов. Римляне украли свой алфавит у греков (римляне украли почти все). Но римляне в действительности не освоили богатства всего греческого языка. Они поленились урвать несколько звуков, потому как не думали, что им понадобятся: Θ (тета), U, V, X, Y и Z. В конечном счете они поняли, что эти звуки также важны, поэтому они добавили их в конец своего алфавита в том порядке, в котором эти звуки были приняты. (Тета никогда не добавлялась римлянами, хотя в некоторых среднеанглийских манускриптах символ Y обозначал эту букву. Именно поэтому, например, “Ye Old Shop” со среднеанглийского переводится как “The Old Shop.”) Это объясняет, как был упорядочен алфавит, т.е. как возник алфавитный порядок. А вот код ASCII появился почти сразу после изобретения телетайпа как способ кодировать числа, обычные символы и секретные коды. Конечно, история возникновения кода ASCII не менее интересна, но это уже другая история, а мы должны заняться программированием.

Еще один пример программы: меню

Чаще всего команда `if` используется для того, чтобы сравнить введенный символ с заранее известным символом и получить ответ типа “да или нет”. Вот пример:

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

```
int main() // главная функция
{
    char c; // символ c;

    // Хотели бы вы, чтобы ваш компьютер взорвался?
    printf("Would you like your computer to explode?");
    c=getchar();
    if(c=='N') // если (c == 'N')
    {
        printf("Okay. Whew!\n"); // Хорошо. Гмм!
    }
    else // иначе
    {
        // ОК: Конфигурирование компьютера для взрыва
    }
}
```

```

        printf("OK: Configuring computer to explode now.\n");
        printf("Bye!\n");
    }
    return(0);
}

```

О! Такая шутка!

Введите предыдущий исходный текст. Сохраните его на диске под именем BLOWUP1.C.

Скомпилируйте и выполните. Вот что вы увидите:

Would you like your computer to explode?

(Хотели бы вы, чтобы ваш компьютер взорвался?)

Благодаря вашему длительному (в течение нескольких лет) обучению компьютерным премудростям, вы знаете, что правильный ответ на этот вопрос — *No* (Нет). Что же может напечатать пользователь? Да Бог знает что! Как программист вы знаете, что нужно напечатать прописную (большую) букву N, чтобы компьютер не взорвался. Любая другая клавиша — и вот вам: "Бум"! Напечатайте N и нажмите <Enter>:

Okay. Whew!

(Хорошо. Гмм!)

Выполните программу снова и нажмите любую клавишу, отличную от N. (Подойдет даже строчная (маленькая) буква n.) Бум! Все, компьютера больше нет!



- ✓ Компьютер не взрывается только в том случае, если пользователь напечатает прописную букву N.
- ✓ К счастью, компьютер не взрывается, — это вы должны были уже выяснить к настоящему моменту.
- ✓ Чтобы сравнить односимвольную переменную с одним символом, все равно нужно два знака "=".
- ✓ Да, эта программа имеет недостатки. Большинство из них описано в главе 14 "Логические выражения и ключевое слово if".

Как с помощью ключевого слова *if* сравнить две строки

Ключевое слово *if* не может сравнивать строки. Оно может сравнить только односимвольные переменные.



Если попытаетесь сравнить две строки с помощью ключевого слова *if*, результат будет непредсказуемый. Программа скомпилируется без каких-либо ошибок (возможно), но она определенно не будет выполняться так, как вы ожидаете.

Логические выражения и ключевое слово if

В этой главе...

- Усовершенствование программ
- Использование логических выражений в условном операторе
- Логическая операция ИЛИ (OR), ||
- Логическая операция И (AND), &&

Команда if умест больше, чем вы, возможно, видели в предыдущих главах этой книги. Однако ее дополнительные возможности связаны с логическими выражениями.

Большинство людей совсем не пугается логики, не то что математики. Дело все в том, что люди не имеют даже самого туманного представления о том, какая логика на самом деле. Если бы они имели хоть малейшее представление о ней, никто никогда не решился бы сыграть в лотерею или посетить Лас-Вегас. Дело в том, что нелогично предположить, что шанс 1 из 80 миллионов что-нибудь да значит. Но я отступаю от темы.

В компьютерных программах встречаются два типа логических выражений. Сначала логические выражения используются для надлежащего принятия решения с помощью ключевого слова if: в таких выражениях используются знаки < или >= и другие глупые вещи, подобные им. Но, кроме того, есть еще логические операции ИЛИ и И и их различные комбинации. Достаточно, чтобы свести вас с ума! Эта глава предотвращает безумие.

Устраняем недостатки в логике программ

Иногда программисты настолько увлекаются тем, работают ли их программы, что они забывают об использовании программ.

Однажды решено было изучить “применимость” конкретной программы. Эта программа задавала пользователям жизненно важный вопрос, и они должны были отвечать, нажимая клавишу табуляции <Tab>. Обычно пользователи ожидают, что нужно нажимать клавишу <Enter>, но при нажатии клавиши <Enter> программа работала необычным образом, и пользователям приходилось отменять три или четыре шага и они возвращались к первоначальному вопросу.

Странная вещь — пользователи имели тенденцию обвинять себя в нажатии <Enter> вместо <Tab>. Они обвиняли себя, но даже не предполагали, что программа ведет себя нелогично. Если бы программист хоть немного следовал логике, программа требовала бы нажимать клавишу <Enter>, а не клавишу табуляции <Tab>. Хотя как программист вы еще только подаете надежды, даже вы знаете, что это действительно просто изменить.

Потратьте одну минуту, чтобы еще раз восстановить в памяти поток управления в программе BLOWUP1.C приведенной в главе 13 “Сравнение символов с помощью ключевого слова if”. Программа задает жизненно важный вопрос:

would you like your computer to explode?

(Хотели бы вы, чтобы ваш компьютер взорвался?)

Конечно, в большинстве случаев пользователи *не хотели бы*, чтобы их компьютер взорвался. Все же программа была написана так, чтобы только нажатие клавиши N — одной клавиши из многих возможных клавиш — препятствовало взрыву компьютера. Это замечательный пример неправильного употребления логического выражения.

Предоставляя пользователю выбор из нескольких опций, нужно сделать так, чтобы пользователю было очень просто (даже при нажатии случайной клавиши) выбрать наилучшую опцию. В случае программы BLOWUP1.C, код, вероятно, должен выглядеть следующим образом:

```
// Хотели бы вы, чтобы ваш компьютер взорвался?
printf("Would you like your computer to explode?");
c=getchar();
if(c=='Y')                                // если (c == 'Y')
{ // OK: Конфигурирование компьютера для взрыва
  printf("OK: Configuring computer to explode now.\n");
}
else                                     // иначе
{
  printf("Okay. Whew!\n");              // Хорошо. Гм!
}
```

Единственная опция в этом примере, которая действительно приводит к взрыву компьютера, — нажатие клавиши Y. Нажатие любой другой клавиши позволяет избежать взрыва компьютера — лучшая опция.

Конечно, выбор лучшей опции должен происходить при вводе вполне определенной информации. Чтобы это сделать, необходимо знать немного больше об уловках команды if, которые представлены в следующем разделе.

- ✓ Вы, возможно, заметили эту нелогичность в главе 13 “Сравнение символов с помощью ключевого слова if”, когда выполнили программу BLOWUP1. Нажимая даже клавишу N для ввода строчной буквы n, вы получаете сообщение о взрыве.
- ✓ На большинство вопросов, которые требуют ответа Yes (Да) или No (Нет), можно ответить, нажав клавишу Y или N, а нажатия любых других клавиш (кроме, возможно, <Esc>) игнорируются.

Команда if и логические операции И (And) и ИЛИ (Or)

С программой BLOWUP1.C связана еще одна интересная проблема. Нет, не как взорвать компьютер. Это просто! Я подразумеваю не эту проблему, а проблему, связанную с вводом строчной буквы y при нажатии клавиши Y!

В предыдущем разделе программа была написана так, что для взрыва компьютера при выполнении программы пользователь должен был нажать только клавишу Y. Но как убедиться, что нажата прописная буква Y, а не строчная y? Или наоборот: что будет, если проверяется только ввод строчной буквы y, а не прописной Y?

Решение (но не лучшее)

Ниже приведен измененный исходный текст BLOWUP1.C:

```
#include <stdio.h>

int main() // главная функция
{
    char c; // символ c;
    // Хотели бы вы, чтобы ваш компьютер взорвался?
    printf("Would you like your computer to explode?");
    c=getchar();
    if(c=='Y') // если (c == 'Y')
    // OK: Конфигурирование компьютера для взрыва
    {
        printf("OK: Configuring computer to explode now.\n");
        printf("Bye!\n");
    }
    else if(c=='y') // иначе если (c == 'y')
    {
    // OK: Конфигурирование компьютера для взрыва
        printf("OK: Configuring computer to explode now.\n");
        printf("Bye!\n");
    }
    else // иначе
    {
        printf("Okay. Whew!\n"); // Хорошо. Гм!
    }
    return(0);
}
```

Отредактируйте первоначальный исходный текст BLOWUP1.C и сделайте изменения, показанные в этом примере. Прежде всего вы проверяете, была ли введена прописная буква Y, а затем добавляется блок else-if, чтобы проверить, не была ли введена строчная буква y (она ведь тоже вводится с помощью клавиши Y). Все инструкции printf() из первого примера программы пришлось повторить, чтобы улучшить логику программы.

Сохраните измененный исходный текст на диске под именем BLOWUP2.C.

Скомпилируйте и выполните.

Если вы напечатаете что-либо, отличное от Y или y, компьютер будет в безопасности. Но если вы напечатаете Y или y, компьютер взорвется (возможно).

- ✓ Зачем проверять и верхний, и нижний регистр? Логика! Вы никогда не знаете, включил ли пользователь режим <Caps Lock>.
- ✓ Поскольку программа проверяет, была ли нажата клавиша Y/y, нажатие любой другой клавиши рассматривается как выбор ответа No (Нет). Это мудро. И даже логично.
- ✓ Более усовершенствованные программы указывают пользователям возможные варианты ответа. Например, в следующей модификации указано, какие клавиши можно нажать в ответ на вопрос:

```
// Хотели бы вы, чтобы ваш компьютер взорвался?
printf("Would you like your computer to explode? (Y/N)");
```

Эта строка указывает, что пользователь может нажать Y или N. (Замечательно, хотя программа определенно не проверяет, была ли нажата клавиша N, — это просто не нужно!)

Логические выражения — лучшее решение

Я не думаю, что есть абсолютно правильный или неправильный способ программировать компьютер, но есть способы более эффективные и менее эффективные. Например, в исходном тексте BLOWUP2.C несколько инструкций повторяются:

```
{
// OK: Конфигурирование компьютера для взрыва
printf("OK: Configuring computer to explode now.\n");
printf("Bye!\n");
}
```

Всякий раз, когда в программе встречается повторение (копия) инструкций, подобное приведенному выше, это обычно означает, что есть лучший способ сделать то же самое. В данном случае проблема связана с двумя командами `if`: одна проверяет, ввел ли пользователь 'Y', а другая — ввел ли пользователь 'y'. Лучше в таком случае использовать логический оператор ИЛИ (OR), чтобы команда `if` имела следующий смысл:

Если переменная `a` равна 'Y' или если переменная `a` равна 'y'

Это на некоторых языках, близких к языкам программирования (на многих псевдокодах, например), записывается так:

```
if(a=='Y' OR a=='y') // если (a == 'Y' ИЛИ a == 'y')
```

Но в C логический оператор для ИЛИ (OR) изображается двумя символами вертикальной черты: `||`. Поэтому данное условие в C записывается так:

```
if(a=='Y' || a=='y') // если (a == 'Y' ИЛИ a == 'y')
```

Эта строка читается так: "Если переменная `a` равна 'Y' или если переменная `a` равна 'y'". Если хотя бы одно из этих высказываний истинно, выполняется набор инструкций, принадлежащих команде `if`.

Ниже приведен модифицированный исходный текст:

```
#include <stdio.h>

int main() // главная функция
{
    char c; // символ c;

    // Хотели бы вы, чтобы ваш компьютер взорвался?
    printf("Would you like your computer to explode?");
    c=getchar();
    if(c=='Y' || c=='y') // если (c == 'Y' || c == 'y')
    {
        // OK: Конфигурирование компьютера для взрыва
        printf("OK: Configuring computer to explode now.\n");
        printf("Bye!\n");
    }
    else // иначе
    {
        printf("Okay. Whew!\n"); // Хорошо. Гм!
    }
    return(0);
}
```

Главная модификация состоит в изменении условия команды `if`, в котором теперь выполняется две операции сравнения и логическая операция ИЛИ (OR). Это позволяет избавиться от избыточных операторов программы.

Сохраните измененный файл на диске под именем BLOWUP3.C. Скомпилируйте. Исправьте все ошибки. Затем выполните полученную программу.

Не правда ли, она стала вдвое логичнее?!

- ✓ Символ `|` называется также *вертикальной чертой*. Он находится на той же самой клавише, что и символ наклонной черты влево, обычно эта клавиша расположена на основной части клавиатуры справа выше клавиши `<Enter>` (Ввод) или `<RETURN>`.
- ✓ Иногда символ `|` имеет разрыв посередине, а иногда это сплошная линия.
- ✓ На языке C два символа `||` означают ИЛИ (OR) — логический оператор, выполняемый над двумя условиями.
- ✓ Условия нужно записывать с обеих сторон логического оператора ИЛИ (OR) `||`.
- ✓ Если условие команды `if` представляет собой результат, возвращаемый логическим оператором `||`, то условие будет истинным, если истинным является хотя бы один операнд логического оператора `||`. В каждом условии — операнде — можно использовать операции сравнения, указываемые операторами, перечисленными в табл. 12.1 главы 12 “Могущественная команда `if`”.

Логические друзья команды `if`

Чтобы с помощью команды `if` в одной инструкции сделать выбор из нескольких альтернатив, для принятия решения вы можете использовать логические операторы `&&` и `||`.

`&&` — логический оператор AND (И).

`||` — логический оператор OR (ИЛИ).

Оба они объясняются в табл. 14.1.

Таблица 14.1. Логические операторы, используемые в условиях команды `if`

Оператор	Смысл	Примеры выражений, принимающих значение <i>true</i> (ИСТИНА)
<code> </code>	OR (ИЛИ)	<code>true true</code> (ИСТИНА ИСТИНА) <code>true false</code> (ИСТИНА ЛОЖЬ) <code>false true</code> (ЛОЖЬ ИСТИНА)
<code>&&</code>	AND (И)	<code>true true</code> (ИСТИНА && ИСТИНА)

Логический оператор обычно соединяет два результата операций сравнения в команде `if` (если). Например:

```
// если (температура>65 && температура<75)
if(temperature>65 && temperature<75)
{
    // замечательная погода
    printf("My, but it's nice weather outside\n");
}
```

В этом примере команда `if` (если) делает два сравнения: `temperature>65` (температура > 65) и `temperature<75` (температура < 75). Если оба выражения истинны, логическая операция AND (И), обозначенная оператором `&&`, вычисляет условие, оно оказывается также истинным и потому выполняется оператор, стоящий в фигурных скобках: функция `printf()` отображает строку. В табл. 14.2 показано, как вычисляется выражение.

Таблица 14.2. Вычисление значения логической операции AND (И)

<i>temperature</i> (температура)	<i>temperature>65</i> (температура>65)	AND (И)	<i>temperature<75</i> (температура < 75)	Результат логической операции AND (И)
45	45>65 FALSE (ЛОЖЬ)	&&	45<75 TRUE (ИСТИНА)	FALSE (ЛОЖЬ)
72	72>65 TRUE (ИСТИНА)		72<75 TRUE (ИСТИНА)	TRUE (ИСТИНА)
90	90>65 TRUE (ИСТИНА)	&&	90<75 FALSE (ЛОЖЬ)	FALSE (ЛОЖЬ)

Согласно таблицам 14.1 и 14.2 оба условия, вычисляемые командой `if`, должны быть истинными, чтобы в результате логической операции AND (И) получилось значение TRUE (ИСТИНА) — только в этом случае будут работать инструкции, принадлежащие ключевому слову `if`. (В данной программе невозможен случай, когда оба условия принимают значение FALSE (ЛОЖЬ). Если бы такое случилось, в результате логической операции AND (И) получилось бы значение FALSE (ЛОЖЬ).)

Ниже приведен пример применения логической операции OR (ИЛИ), которая используется в условном операторе для проверки двух условий:

```
// если (счет> 100 || cheat_code == 'Y')
if(score>100 || cheat_code=='Y')
{
    // Вы победили босса этого уровня
    printf("You have defeated the level boss!\n");
}
```

Логическая операция OR (ИЛИ) позволяет проверить, является ли истинным хотя бы одно из двух условий, в частности, она позволяет проверить, истинно ли хотя бы одно из двух неравенств. Это часто помогает упростить программу, так как иногда удастся сократить количество команд `if` (если). В табл. 14.3 приведены примеры вычисления результатов логической операции OR (ИЛИ).

Таблица 14.3. Выполнение логической операции OR (ИЛИ)

<i>Score (Счет), cheat_code</i>	<i>score>100 (счет> 100)</i>	OR (ИЛИ)	<i>cheat_code=='Y'</i>	Результат логической операции OR (ИЛИ)
50,Z	50>100 FALSE (ЛОЖЬ)		'Z'=='Y' FALSE (ЛОЖЬ)	FALSE (ЛОЖЬ)
200,Z	200>100 TRUE (ИСТИНА)		'Z'=='Y' FALSE (ЛОЖЬ)	TRUE (ИСТИНА)
50,Y	50>100 FALSE (ЛОЖЬ)		'Y'=='Y' TRUE (ИСТИНА)	TRUE (ИСТИНА)
200,Y	200>100 TRUE (ИСТИНА)		'Y'=='Y' TRUE (ИСТИНА)	TRUE (ИСТИНА)

В табл. 14.3 показано, как логическая операция OR (ИЛИ) использует результаты операций сравнения. Если хотя бы один из них принимает значение TRUE (ИСТИНА), то же значение принимает и все выражение. В отличие от логической операции AND (И), для истинности результата которой требуется истинность обоих (т.е. всех) условий, для истинности результата логической операции OR (ИЛИ) требуется истинность хотя бы только одного неравенства — это гарантированно обеспечит истинность всего условия в условном операторе.

В большинстве случаев логические операторы `&&` и `||` применяются только к двум условиям. Однако в принципе их можно применить сразу к нескольким операндам. Это допускается, хотя может выглядеть неряшливо. Все следующие примеры являются правильными, хотя их вычисление может показаться и не совсем очевидным:

```
// если (клавиша == 'A' || клавиша == 'B' || клавиша == 'C')
if(key=='A' || key=='B' || key=='C')
```

В предыдущей строке условие в условном операторе истинно, если значение символьной переменной `key` (клавиша) равно А или В или С. Любое из этих значений гарантирует истинность всего логического выражения, а, значит, и выполнение инструкций, принадлежащих ключевому слову `if`.

В следующем примере все три условия-операнда должны иметь значение `TRUE` (ИСТИНА), чтобы это значение имело все условие. Значение температуры (`temperature`) должно быть больше 70, значение `sun` (солнце) должно равняться значению `shining` (сиять), а значение `weekend` (уикенд) должно равняться константе `YES` (ДА). Если все три условия истинны, то истинно и все условие инструкции `if`.

```
// если (температура > 70 && солнце == сиять && уикенд == ДА)
if(temperature>70 && sun==shining && weekend==YES)
```

Следующий пример несколько сложнее.

```
// если (состояние == 1 || пользователь == КОРЕНЬ && таймер < МАКСИМУМ)
if(status==1 || user==ROOT && timer<MAX)
```

Логические операции выполняются слева направо. Каждое условие также вычисляется слева направо: `status` (состояние) может быть равно 1 или `user` (пользователь) может быть равен `ROOT` (КОРЕНЬ) — хотя бы одно из этих условий должно быть истинным. Но если `status` не равно 1, а `user` равен `ROOT`, то `timer` (таймер) должен быть меньше, чем `MAX` (МАКСИМУМ), чтобы в данной инструкции было истинным все логическое выражение. Гм!

- ✓ Да, от вычисления логических выражений нагреваются даже процессоры. Если вам жарко, пожалуйста, сделайте перерыв.
- ✓ Легко запомнить, что `&&` обозначает “логический оператор AND (И)”, потому что символ амперсанда обозначает “and (и)” по-английски.
- ✓ Но не так просто запомнить, что `||` обозначает “логический оператор OR (ИЛИ)”.
- ✓ Операнды (условия) операций `&&` и `||` вычисляются слева направо. Любое ложное условие делает ложным все логическое произведение.
- ✓ Есть различие между `&&` и `&` (один амперсанд). Один `&` в С обозначает совсем другую операцию, так что пишите `&&` в логических выражениях в условном операторе. То же самое относится и к `||` и `|`; в условном операторе используйте `||`.
- ✓ `|` и `&` — поразрядные операторы, выполняющие логические операции над битами.
- ✓ Сам амперсанд — сокращение латинского *et*, что означает *и*. Да и символ `&` — просто стилизованная комбинация *e* и *t*.



Программа, в которой используется логическая операция AND (И)

Следующий исходный текст демонстрирует применение логической операции AND (И). Это еще одна — заключительная — модификация ВЗРЫВАЮЩИХСЯ программ BLOWUP.

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

```
int main() // главная функция
{
    char c,d; // символ c, d;

    // Введите символьный код для самоликвидации
    printf("Enter the character code for self-destruct?");
    c=getchar();
    /* используйте fpurge(stdin) в unix */
    fflush(stdin);
    // Введите код числа, чтобы подтвердить самоликвидацию
    printf("Input number code to confirm self-destruct?");
    d=getchar();
    if(c=='G' && d=='0') // если (c == 'G' && d == '0')
    {
        // АВТОЗАПУСК ДЕСТРУКЦИИ
        printf("AUTO DESTRUCT ENABLED!\n");
        printf("Bye!\n");
    }
    else // иначе
    {
        printf("Okay. Whew!\n"); // Хорошо. Гм!
    }
    return(0);
}
```

Начните редактировать исходный текст BLOWUP3.C, или просто введите этот код в пустой файл. Сохраните конечный результат на диске под именем BLOWUP4.C.

Скомпилируйте. Исправьте все ошибки. Запустите на выполнение.

Все коды вы знаете, поскольку сами вводили исходный текст: большая (прописная) буква G и 0 (нуль). Чтобы взорвать компьютер, их оба нужно ввести с клавиатуры.

- ✓ Эта программа использует команду `fflush(stdin)` для очистки буфера от остатков ввода после первого вызова функции `getchar()`. В главе 13 “Сравнение символов с помощью ключевого слова `if`” подробно описано, почему необходима эта команда.
- ✓ Если вы пользуетесь операционными системами семейства Unix, применяйте `fpurge(stdin)` вместо `fflush(stdin)`. Подробности найдете в главе 13 “Сравнение символов с помощью ключевого слова `if`”.
- ✓ Логический оператор `&&` — знак операции AND (И) — гарантирует, что все условие условного оператора будет истинным только в том случае, если обе переменные с и d имеют надлежащие значения.
- ✓ Вы можете также учесть случай ввода строчной (маленькой) буквы g, если с помощью редактора сделаете следующее изменение в исходном коде:

```
// если ((c == 'G' || c == 'g') && d == '0')
if(c=='G' || c=='g' && d=='0')
```

Сначала проверяется, равна ли переменная с G или g — любая из этих букв является кодом символа. Затем вычисляется результат операции сравнения d с нулем: `d == 0`. (Чтобы взорвать компьютер, переменная d должна быть равна 0.) Если переменная с не равна ни G, ни g, результатом операции OR (ИЛИ) будет FALSE (ЛОЖЬ). В этом случае, а также если результатом последней операции сравнения является FALSE (ЛОЖЬ), все условие в данной инструкции является ложным. Ага, испугался!

Циклы в языке C

В этой главе...

- Понятие цикла
- Повторение части программы с помощью оператора `for`
- Счет с помощью цикла
- Отображение таблицы ASCII с помощью цикла
- Как избежать заикливания
- Прерывание цикла с помощью оператора `break`

Большее всего компьютеры любят повторять одни и те же действия. А люди? О, нет! Нет большего наказания, чем заставить ребенка написать 100 раз на классной доске одну и ту же фразу, например “Не следует хихикать, когда учитель с умным видом делает глупые ошибки”. А компьютеры? Они нисколько не возражают. На самом деле они наслаждаются выполнением подобных заданий.

Наиболее ценным свойством программ является возможность принятия решений, после которого следует возможность повторять действия. Единственная проблема заключается в том, чтобы вовремя их остановить. А для этого, прежде чем перейти к тонкостям выполнения инструкции цикла, нужно знать, как работает инструкция `if`. Эта глава начинается со знакомства с выполнением цикла `for`, наиболее древней из команд, связанных с циклами.

- ✓ Более подробно условный оператор `if` описан в главах 12 “Могущественная команда `if`” и 14 “Логические выражения и ключевое слово `if`”.
- ✓ Полезно просмотреть табл. 12.1 из главы 12 “Могущественная команда `if`”, где сравниваются функции операторов `if` и `for`.

Понятие цикла

Циклом называется многократное выполнение одних и тех же действий. Например, чтобы программа сосчитала от 1, скажем, до миллиона, в ней нужно записать цикл. Цикл — это та часть кода программы, которая выполняется заданное количество раз.

В качестве примера цикла рассмотрим следующие действия, обычно выполняемые маленьким ребенком и мамой в процессе знакомства со столь удивительным предметом, как ложка:

Ребенок берет ложку.

Ребенок бросает ложку на пол; ребенок смеется.

Мама поднимает ложку с пола.

Мама кладет ложку перед ребенком.

Повторение.

Это подобно выполнению некоей примитивной программы. Более того, это можно даже рассматривать как программу, ведь фактически это последовательность инструкций! И в ней есть цикл. Это видно по слову “Повторение”, которое показывает, что определенная последовательность шагов повторяется.

К сожалению, в примитивной предыдущей программе отсутствует условие остановки. Это и есть так называемый бесконечный цикл. Его выполнение приводит к заикливанию. В языках программирования большинство операторов цикла содержат условия, подобные тем, которые используются в условном операторе. Эти условия указывают, когда нужно прекратить повторения. Отсутствие таких условий нежелательно.

Небольшие изменения в программе могут помочь избежать заикливания:

Повторять, пока мама не перестанет класть ложку перед ребенком.

Теперь в программе есть условие остановки цикла.

Цикл имеет три части:

- ✓ начало;
- ✓ средняя часть, которая повторяется;
- ✓ конец.

Итак, всякий цикл состоит из этих трех частей. Начало — это то место, где устанавливаются начальные значения, используемые для запуска цикла, обычно это некоторая команда языка программирования, которая говорит что-нибудь вроде следующего: “здесь начинается цикл — надо будет что-то повторять”. Средняя часть состоит из команд, которые повторяются много раз. Конец отмечает завершение повторяющейся части или условие окончания цикла, программисты его часто называют условием выхода из цикла. (В нашем примере: пока мама не перестанет класть ложку перед ребенком).

- ✓ В языке программирования C предусмотрено несколько различных типов циклов. Так, в языке программирования C предусмотрен не только цикл `for`, который рассматривается в данной главе, но и цикл с условием продолжения `while`, а также цикл с условием продолжения `do-while`. Кроме того, цикл можно организовать также с помощью уродливого ключевого слова `goto`, использовать которое не рекомендуется. Конечно, не исключено, что оно может встретиться где-нибудь в этой книге.
- ✓ Команды внутри цикла выполняются определенное количество раз или же до тех пор, пока не будет выполнено условие выхода из цикла. Например, можно сказать компьютеру: “выполняй это много раз” или же “выполняй это, пока бегунок не дойдет до конца”. В любом случае последовательность операций будет выполняться много раз.
- ✓ После выполнения (повторения) цикла выполнение программы будет продолжено. Но пока цикл не закончится, много раз повторяется выполнение одной и той же части программы.

Циклическое повторение без проблем

Ниже представлен исходный текст программы OUCH.C. Эта программа показывает, что компьютеру не составляет труда повторять выполнение одних и тех же команд много раз. Делает он это с удовольствием, без каких бы то ни было жалоб, причем так быстро, что вы даже не успеете не то что поджарить кукурузу, но даже открыть коробку с попкорном. (Я не говорю уже о том, чтобы устроить себе банкет с газированными напитками.)

В данной программе используется ключевое слово `for`. На языке C с помощью этого ключевого слова записывается одна из самых основных команд, которая заставляет компьютер выполнять цикл. В данной программе ключевое слово `for` позволяет создать небольшой цикл, в котором пять раз повторяется единственная команда `printf()`.

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

```
int main()                                // главная функция
{
    int i;

    for(i=0 ; i<5 ; i=i+1)
    {
        // Ой! Пожалуйста, остановитесь!
        printf("Ouch! Please, stop!\n");
    }
    return(0);
}
```

Введите этот исходный текст с помощью вашего редактора.

В этой программе встречается новое ключевое слово `for`. Тщательно перепроверьте текст. Обратите внимание, что строка, которая начинается с `for`, не заканчивается точкой с запятой. Вместо этого ставятся фигурные скобки, точно так же, как и в условном операторе. (Фактически цикл выглядит точно так, как условный оператор, если игнорировать отличие в ключевых словах.) Сохраните файл под именем `OUCH.C`.

Скомпилируйте файл `OUCH.C`, используя ваш компилятор. Исправьте все ошибки. Не пропустите точки с запятой в скобках ключевого слова `for`.

Выполните полученную программу. Вы увидите на экране следующее:

```
Ouch! Please, stop!
Ouch! Please, stop!
Ouch! Please, stop!
Ouch! Please, stop!
Ouch! Please, stop!
```

(Ой! Пожалуйста, остановитесь!

Ой! Пожалуйста, остановитесь!

Ой! Пожалуйста, остановитесь!

Ой! Пожалуйста, остановитесь!)

Теперь убедились, что повторение не вредит компьютеру? Ни капельки!

- ✓ Цикл `for` имеет начало, середину и конец. Средняя часть — инструкция `printf()` — это та часть, которая повторяется. Остальная часть цикла, начало и конец, находится в круглых скобках ключевого слова `for`. (В следующем разделе выполнение цикла рассматривается более подробно.)

- ✓ Как и в случае с конструкцией `if`, когда только одна инструкция принадлежит команде `for`, все можно записать так:

```
for(i=0 ; i<5 ; i=i+1)
    // Ой! Пожалуйста, остановитесь!
    printf("Ouch! Please, stop!\n");
```

Фигурные скобки не обязательны, если нужно повторять только одну инструкцию. Но они необходимы, если требуется повторять несколько инструкций.

- ✓ В цикле `for` инструкция `printf()` повторяется пять раз.



✓ В цикле `for` скрыта следующая инструкция:

`i=i+1`

В языке программирования C так обозначается добавление 1 к переменной. Это действие называется *приращением*, и вы должны прочитать главу 11 “Больше математики и Священный порядок (старшинство операций)”, если вы не знаете, что это такое.

✓ Не стоит волноваться, если вы все еще не можете понять суть ключевого слова `for`. Надо много раз прочитать всю эту главу, постоянно бормоча “я пойму этот материал”. Затем при необходимости перечитайте все снова, причем начните с самого начала. Но сперва прочитайте всю главу до конца.

Для того чтобы сделать что-нибудь много раз, используйте ключевое слово `for`

Слово `for` является одним из тех слов, которое становится все более и более сверхъестественным и чудодейственным, чем больше вы произносите его; во всяком случае, в английском языке оно может означать для (*for*), поскольку (*for*), передний (*fore*), четыре (*four*), холл (*foyer*)... Как показывают следующие примеры, оно даже может быть ключевым в грамматической структуре предложения:

For he's a jolly good fellow. (Поскольку он — веселый хороший товарищ.)

These are **for** your brother. (Эти штуковины для твоего брата.)

For why did you bring me here? (Зачем вы привели меня сюда?)

An eye **for** an eye. (Око за око.)

For the better to eat you with, my dear. (Чтобы быстрее тебя съесть, моя дорогая.)

Three **for** a dollar. (Три за доллар.)

И так далее. Но, пожалуй, лучше я прекращу перечислять примеры с этим словом. Потому что оно может раздражать. (**For** it can be maddening.)

В языке программирования C ключевое слово `for` используется для организации цикла, который, естественно, называется циклом `for`. Ключевое слово `for` определяет начальное состояние, условие окончания и команды, которые выполняются много раз во время повторения цикла. Формат может показаться немного сложным, поэтому сначала рассмотрим несколько упрощенный синтаксис цикла `for`.

for(начало; условие; приращение) // для(начало; условие; приращение)
инструкция;

После ключевого слова `for` открываются круглые скобки. В круглых скобках содержится три элемента цикла, отделенные двумя точками с запятой. Точка с запятой в конце строки, содержащей слово `for`, не ставится.

Первый элемент *начало* устанавливает начальное состояние цикла. Элемент *начало* обычно представляет собой инициализацию некоторой переменной ее начальным значением. В OUCH.C *начало* обычно представляет собой инструкцию `i=0`.

Второй элемент — *условие*. Цикл продолжает выполняться до тех пор, пока истинно условие, указанное вторым элементом цикла. (Иными словами, повторения продолжаются до тех пор, пока выполняется условие, указанное вторым элементом цикла.) Как правило, *условие* содержит операцию сравнения; оно записывается точно так же, как и условие в команде `if`. В OUCH.C это `i<5`.

Последний элемент, содержащийся в скобках ключевого слова `for` — *приращение*. В этом элементе указывается, что нужно сделать после очередного выполнения цикла. В OUCH.C на этот элемент возложена задача увеличения переменной `i` на 1: `i=i+1`.

Элемент *инструкция* представляет собой оператор. Он следует после ключевого слова `for`. (Иногда даже говорят, что *инструкция* принадлежит ключевому слову `for`.) Выполнение *инструкции* повторяется заданное число раз — до тех пор, пока продолжается выполнение цикла, заданного ключевым словом `for`. Эта *инструкция* должна заканчиваться точкой с запятой.

Если конструкции `for` принадлежит больше одной инструкции, вы должны загнать эти инструкции в фигурные скобки. Вот так:

```
for(начало; условие; приращение)
{
    инструкция;
    инструкция;
    /* и т.д. */
}
```

Обратите внимание, что инструкции являются дополнительными. Вы можете написать цикл `for` (для) и так:

```
for(начало; условие; приращение)
;
```

В этом случае единственная точка с запятой как раз и будет той “инструкцией”, которая предназначена для повторения.

- ✓ Одна из самых больших трудностей, связанных с ключевым словом `for`, состоит в том, чтобы понять, что происходит с тремя элементами, заключенными в круглые скобки ключевого слова `for`. Честно говоря, этого не понимают не только новички, но и знатоки языка C, хотя они и не признаются в этом.
- ✓ Какая самая большая ошибка связана с циклом `for`? Использование запятых, а не точек с запятой. Внутри круглых скобок элементы разделяются точками с запятой!
- ✓ Некоторые программисты используют гибкость языка C и пропускают части *начало* и *приращение* в команде `for`. Хотя это вполне законно, я не рекомендую такой прием для новичков. Просто не падайте в обморок, если вы когда-либо увидите такое. Помните, что внутри круглых скобок ключевого слова `for` всегда должны быть обе точки с запятой.

Пошаговое выполнение программы OUCH.C

Я признаю, что цикл `for` — не самая простая команда языка C, предназначенная для выполнения циклов. Основная трудность состоит в понимании его частей. Конечно, было бы проще написать следующее

```
for(6)
{
    /* инструкции */
}
```

и затем повторить шесть раз инструкции, заключенные в фигурные скобки. Но такой вариант оператора `for` значительно уменьшает его возможности управления инструкциями.

Чтобы лучше понять, какой смысл имеют три части в круглых скобках цикла `for`, в качестве примера рассмотрим следующую инструкцию цикла `for` из программы OUCH.C:

```
for(i=0 ; i<5 ; i=i+1)
```

В цикле `for` переменная `i` используется для того, чтобы сосчитать количество повторений инструкции в этом цикле.

Первый элемент цикла `for` определяет начальное состояние. В данной строке программы начальное значение целочисленной переменной `i` устанавливается равным 0. Эта простая старая инструкция языка C присваивает переменной значение:

```
i=0
```

Внутри круглых скобок значение 0 проскальзывает в целую переменную `i`. Ничего больше не происходит.

Второй элемент — условие. Оно подобно тем условиям, которые используются в условных операторах. Условие представляет собой *условие продолжения выполнения* цикла `for`; цикл может продолжать повторять выполнение своих инструкций много раз, но только до тех пор, пока указанное в нем условие истинно. В предыдущем коде цикл продолжает повторяться до тех пор, пока истинно условие цикла, т.е. пока в результате вычисления инструкции `i<5` (значение переменной `i` меньше 5) получается логическое значение `true`. Условие здесь то же самое, что и в следующем условном операторе:

```
if (i<5)
```

Если значение переменной `i` меньше 5, продолжается выполнение цикла.

Последний элемент цикла `for` указывает, что сделать перед очередным повторением цикла. Без этого элемента цикл повторялся бы вечно: `i` равно 0, а цикл повторяется, пока `i<5` (значение `i` меньше 5). Это условие истинно, так что цикл `for` продолжался бы бесконечно, подобно федеральной программе субсидирования фермеров. Однако последний элемент цикла `for` увеличивает значение переменной `i` в конце очередного повторения цикла:

```
i=i+1
```

Компилятор берет значение переменной `i` и добавляет к нему 1 каждый раз, когда завершается очередное повторение цикла `for`. (Подробнее операция приращения рассмотрена в главе 11 “Больше математики и Священный порядок (старшинство операций)”.) Таким образом, циклу `for` удастся повторить себя — и все инструкции, которые принадлежат ему — всего пять раз. В табл. 15.1 подробно описано выполнение цикла.

Таблица 15.1. Изменение значения переменной `i` в цикле `for`

Значение <code>i</code>	Выполнено ли условие <code>i<5</code> ?	Инструкция	Что сделать
<code>i=0</code>	Да, продолжать выполнение цикла ←	<code>printf()...</code> (1)	<code>i=0+1</code>
<code>i=1</code>	Да, продолжать выполнение цикла ←	<code>printf()...</code> (2)	<code>i=1+1</code>
<code>i=2</code>	Да, продолжать выполнение цикла ←	<code>printf()...</code> (3)	<code>i=2+1</code>
<code>i=3</code>	Да, продолжать выполнение цикла ←	<code>printf()...</code> (4)	<code>i=3+1</code>
<code>i=4</code>	Да, продолжать выполнение цикла ←	<code>printf()...</code> (5)	<code>i=4+1</code>
<code>i=5</code>	Нет — остановиться немедленно!		

В начале табл. 15.1 значение переменной `i` равно 0, т.е. начальному значению, заданному в инструкции `for`. Затем вычисляется второй элемент — выполняется операция сравнения, т.е. проверяется условие. Верно ли неравенство `i<5`? Если да, цикл продолжается.

При выполнении цикла вычисляется третья часть инструкции `for`, и значение `i` увеличивается. Наряду с этим, выполняются любые инструкции, принадлежащие команде `for`. Когда выполнение всех этих инструкций закончено, снова выполняется операция сравнения `i<5` и цикл либо повторяется, либо останавливается — в зависимости от результата этой операции сравнения.



- ✓ Цикл `for` может быть громоздким, ведь он имеет так много частей! Поэтому в нем иногда трудно разобраться. Но, тем не менее, в языке программирования C это самый удобный способ записать повторное выполнение группы инструкций заданное количество раз.
- ✓ Третий элемент в круглых скобках инструкции `for` — *приращение* — выполняется только один раз при каждом повторении цикла. И это утверждение истинно независимо от количества инструкций, принадлежащих циклу, — их может быть одна или несколько или даже они могут отсутствовать вообще (в этом случае их количество равно нулю).
- ✓ Большинство новичков, знакомясь с циклом `for`, испытывают трудности, стремясь понять назначение второго элемента. Они помнят, что первый элемент означает “здесь начало цикла”, но иногда они думают, что второй элемент означает “здесь конец цикла”. Это не так! Второй элемент означает “продолжать выполнение цикла, пока заданное условие истинно”. Этим второй элемент подобен операции сравнения в `if`. Компилятор, конечно, не подозревает о таких ляпах, но программа работает не так, как ожидает новичок. В результате, когда новичок делает эту ошибку, он думает, что программа не работает должным образом, хотя на самом деле ошибочно его представление о том, как она должна работать.



- ✓ Не забывайте объявить переменную, используемую в круглых скобках цикла `for`. Эту частую ошибку допускают почти все. Подробно объявление переменных рассмотрено в главе 8 “Переменные в языке C”.
- ✓ Ниже в виде цикла `for` приведена заготовка, которую удобно использовать в программах. В следующей строке нужно только заменить прописную (большую) букву X числом — количеством выполнений цикла:

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

Вы должны объявить, что `i` будет переменной целочисленного типа. Начальное ее значение равно 1, а конечное — значению X. Например, чтобы повторить цикл 100 раз, напишите следующую команду:

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

Забава: считаем до 100

В этом разделе представлен исходный текст для программы 100.C. Эта программа использует цикл `for`, чтобы посчитать до 100 и отобразить каждое число на экране. И действительно, это — большое достижение: первые компьютеры могли считать только до 50, а затем они начали делать дикие и часто ошибочные предположения о том, какое число должно быть следующим. (Посмотрите на ваш телефонный счет, и вы сразу поймете, что я имею в виду.) Основная часть программы 100.C подобна OUCH.C. На самом деле здесь этот пример приведен только потому, что цикл `for` многим новичкам кажется настолько странным, что одной программы мало для разъяснения всех тонкостей. Поэтому-то мы переходим к рассмотрению следующего примера программы:

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

```
int main()
```

```
// главная функция
```

```
{
```

```
    int i;
```

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

```
printf("%d\t", i);  
return(0);  
}
```

Введите этот исходный текст с помощью вашего редактора. Соблюдайте отступы; в лучших традициях языка С даже когда фигурные скобки опущены, нужно делать отступ, если одна инструкция принадлежит другой (как показано в примере).

В инструкции `for` переменной `i` сначала присваивается 1. Условие (второй элемент инструкции `for`) — `i<=100`. Благодаря этому условию цикл будет повторяться, пока значение `i` меньше или равно 100. Заключительная часть инструкции увеличивает значение `i` на 1 в конце каждого повторения цикла.

Инструкция `printf` отображает значение целой переменной `i`, используя метку-заполнитель `%d`. Escape-последовательность `\t` вставляет символ табуляции в вывод, выстраивая все выводимые числа в красивые, ровные, аккуратные столбцы.

Сохраните файл на диске под именем 100.C. (Это имя должно вам дважды напоминать о назначении программы, ведь римская цифра С — это 100. Вот так.)

Скомпилируйте программу и выполните ее. Вы увидите на экране числа от 1 до 100, расположенные в десяти аккуратных строках и десяти столбцах. Просто удивительно, как быстро компьютер может сделать это.

✓ При каждом из 100 повторений инструкции `printf()` в цикле `for` отображается значение переменной `i` и увеличивается значение этой переменной. (Таким образом, увеличение выполняется 100 раз.)

✓ Измените исходный текст 100.C. В инструкции `for` в заголовке цикла измените количество повторений так, чтобы цикл выполнялся 10 000 раз. Для этого с помощью редактора отредактируйте заголовок цикла так:

```
for(i=1 ; i<=10000 ; i=i+1)
```

Достаточно лишь вставить два дополнительных нуля после числа 100, которое уже имеется в заголовке цикла. Сохраните измененный файл на диске и перетранслируйте его. Компьютеру потребуется не на много больше времени, чтобы сосчитать до 10 000; однако ему понадобится гораздо больше времени, чтобы отобразить все числа от 1 до 10 000.

Поговорим о циклах!

Цикл — это одно из самых полезных средств программирования. Циклы в программах — это как роскошные, фантастические, достойные самых высоких призов сливочные украшения на вершинах холодных унылых гор салата из хрустящей капусты. Исключительно благодаря циклам компьютеры могут выполнять полезные действия. Поэтому не удивительно, что без цикла не обходится ни одна из следующих операций: сортировка, поиск, распечатка, организация поддержки. И, конечно же, ни в одной полезной программе нельзя обойтись без цикла. В циклах — соль программирования.

Я откладываю полный обзор применения циклов до главы 17 “Познакомьтесь с циклом `while` (циклом с условием продолжения)”. Моя задача в этом разделе состоит в том, чтобы завершить это краткое (но все же несколько преждевременное) описание замечательного ключевого слова `for` и показать вам некоторые интересные средства — я называю их антициклами. — предназначенные для прерывания даже самой (сверх) прилежной компьютерной программы.

Полезная программа ASCII

Наконец, в этом разделе я приведу исходный текст программы ASCII.C. Несомненно, это первая полезная программа в книге. Программа ASCII.C отображает символы и их соответствующие коды из таблицы ASCII, начиная от кода 32 и до кода 127. Эта программа полезна потому, что из подсказки DOS вы можете легко напечатать таблицу ASCII, и вам не нужно будет постоянно искать коды в приложениях самых разных книг. (Программисту приходится искать коды ASCII почти ежедневно.)

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

```
int main()                                // главная функция
{
    unsigned char a;                      // символ без знака a;

    for(a=32;a<128;a=a+1)
        printf("%3d = '%c'\t",a,a);
    return(0);
}
```

Введите этот исходный текст с помощью вашего редактора. Этот исходный текст содержит основной цикл `for`, который повторяет инструкцию `printf()` несколько десятков раз.

После перепроверки исходного текста сохраните его на диске под именем ASCII.C.

С помощью компилятора скомпилируйте программу.

После запуска программы на экране вы увидите 5 столбцов, в которых будут отображены символы и их ASCII-коды, начиная с кода 32 (символ пробела) и заканчивая кодом 127, который выглядит как маленький домик, хотя на самом деле является греческой буквой дельта Δ.

- ✓ Эту программу я использую для быстрого поиска ASCII-кодов.
- ✓ В программе ASCII.C в цикле `for` начальное значение переменной `a` равно 32. Затем значение переменной `a` увеличивается (на 1) до значения 127 — последнего значения, при котором выполняется оператор печати, поскольку `a<128`. (После этого условие `a<128` выполнено не будет, и цикл остановится.) Приращение удобно выполнить с помощью инструкции `a=a+1`, расположив ее в круглых скобках ключевого слова `for`.
- ✓ Стоит отметить, что в функции `printf()` используется символьная переменная `a` дважды (сначала как целое число, а потом — как символ). Однако чтобы не писать инструкцию вывода дважды, в строке формата для `printf()` используются метки-заполнители `%d` и `%c`.
- ✓ Число 3 в метке-заполнителе `%3d` форматной строки функции `printf()` указывает, что `printf()` должна всегда отображать три символа, когда печатает целочисленное значение. Если ширину поля, используемого для отображения значения, установить равной трем символам, все значения (коды) будут выстроены в столбик, выровненный по правому краю. Благодаря этому все знаки разряда единиц окажутся друг под другом (поскольку перед двухцифровыми числами будет вставлен пробел).
- ✓ Вот теперь пришло время открыть вам страшную тайну: переменная `a` в ASCII.C может рассматриваться и как целое число, и как символьная переменная. В программе ее можно объявить как имеющую любой из этих типов. Дело в том, что в цикле `for` значение переменной `a` используется как целое число, а в функции





`printf()` это же значение используется и как символ, и как число. Эта двойственность справедлива и для типа `int` (для целых чисел), и для типа `char` (для символов), однако лишь для тех переменных, значения которых не превосходят 255 (это наибольшее “значение”, которое можно хранить в символьной (типа `char`) переменной).

- ✓ Вот еще одна страшная тайна: переменная должна быть объявлена как символ без знака (`unsigned char`). Если в программе переменная объявлена как `unsigned` (без знака), ее значения не могут быть отрицательными числами. Если бы переменная была объявлена только как символьная переменная (типа `char`), цикл повторялся бы бесконечно; сложение 1 с `a`, когда значение `a` равняется 127, даст значение -127, и цикл просто повторялся бы вечно. (Чтобы доказать это, вырежьте слово `unsigned` (без знака) в исходном тексте ASCII.C, и перетранслируйте программу, после этого она будет выполняться вечно (без останова), а это совсем не то, что вам надо.)
- ✓ Разговор о выполнении цикла можно вести в цикле (бесконечном)...

Остерегайтесь бесконечных циклов!

Некоторые вещи вечны. Говорят, Любовь вечна. Бриллианты, конечно. Смерть и налоги — да, они тоже, к сожалению. Некоторые циклы также могут быть вечны, даже если вы не хотите, чтобы они были вечными. “Вечный” — слишком философский термин. Капризные программисты предпочитают термин “бесконечный”. “Бесконечный” значит “продолжающийся вечно и никогда не останавливающийся”. Да, это что-нибудь вроде кролика из рекламы “Energizer”.

Бесконечный цикл — это повторяющийся раздел программы, притом повторяющийся бесконечно. Я не могу придумать никакого практического применения для такого цикла. Фактически бесконечный цикл — обычно несчастный случай в программировании. Это ошибка, которая может произойти при проектировании даже весьма необходимой программы. Часто о ее существовании программист даже не подозревает до выполнения программы. Только когда программа заикнется и ничего другого не делает или когда она снова и снова выплескивает на экран одни и те же символы, не подсказывая, как остановиться, вы понимаете, что вы создали бесконечный цикл. Увы, когда-нибудь такое случается со всеми.

Следующая программа — `FOREVER.C` — нарочно содержит бесконечный цикл. Вы можете ввести эту программу и испытать. Используя обман в команде `for`, программа повторяет инструкцию `printf()` до бесконечности:

```
#include <stdio.h>

int main()                                // главная функция
{
    int i;

    for(i=1;i=5;i=i+1)
        // компьютер взбесился!
        printf("The computer has run amok!\n");
    return(0);
}
```

Введите этот исходный текст с помощью текстового редактора. Эта программа похожа на первую программу в этой главе, содержащую цикл `for`, — на программу `OUCH.C`. Различие находится в цикле `for`, точнее в той его части, которая проверяет необходимость повторения цикла, т.е. в условии. Кроме того, отличается и текст сообщения, отображение которого повторяется. Сохраните исходный текст на диске под именем `FOREVER.C`.

Скомпилируйте программу. Даже при том, что в инструкции `for` преднамеренно содержится бесконечный цикл, никакое сообщение об ошибках не отображается (если вы не попали впросак и напечатали что-нибудь неправильно). В конце концов, компилятор может думать, что вы пытаетесь делать что-то вечно, и эти вечно повторяемые действия являются частью вашего главного плана. Действительно, что бы это еще могло значить?

Когда вы будете выполнять программу, вы увидите, что на экране безумно, без остановки будет пробегать следующее сообщение:

The computer has run amok!

(Компьютер взбесился!)

Действительно, это так и есть! Нажмите комбинацию клавиш `<Ctrl+C>`, чтобы остановить безумие.

- ✓ Большинство циклов проектируются так, чтобы сработало условие окончания цикла. Цикл, который закичивается, либо не имеет условия, либо условие записано так, чтобы цикл не заканчивался. Это плохо.
- ✓ Бесконечные циклы очень коварны! Часто вы не можете обнаружить их до тех пор, пока программа не начнет выполняться. Именно поэтому нужно проверять каждую создаваемую программу.
- ✓ И в Windows, и в Unix для отмены команды печати на стандартное устройство вывода используется комбинация клавиш `<Ctrl+C>`. Как вы помните, команда печати на стандартный вывод выполняется в `FOREVER.C` много раз (бесконечно, если ее не остановить). И, как вы помните, именно комбинация клавиш `<Ctrl+C>` позволила отменить печать на стандартное устройство вывода, притом независимо от того, запустили вы программу в Windows или в Unix. Другие типы программ с бесконечными циклами, особенно те, которые ничего не печатают на стандартное устройство вывода, остановить намного тяжелее. Если `<Ctrl+C>` не помогает, часто приходится прибегать к специальным средствам конкретной операционной системы (например, к команде `kill`), чтобы *убить* взбесившуюся программу.
- ✓ В былые дни часто приходилось полностью перезапускать компьютер, чтобы восстановить контроль над ним, утерянный в результате выполнения взбесившейся программы, закичивание которой произошло из-за выполнения бесконечного цикла.
- ✓ Программа закичивается (выполняет бесконечный цикл) из-за ошибки в условии цикла `for`. Условие, как вы помните, — второй элемент в круглых скобках:

```
for (i=1; i=5; i=i+1)
```

Компилятор языка C видит `i=5` и рассуждает: “Хорошо, я помешу 5 в переменную `i`.” Но это — не операция сравнения, в результате которой получается истина или ложь. Ведь это совсем не похоже на обычное условие в условном операторе, — а именно его там и ожидал компилятор! Поэтому компилятор предполагает, что нужно вычислить записанное на месте условия выражение. В данном случае в результате вычисления условие оказывается истинным и потому выполнение цикла продолжается — фактически независимо ни от чего. Обратите внимание, что переменная `i` всегда равна 5 именно по причине вычисления “условия” `i=5`; ведь даже после того, как она увеличивается с помощью инструкции `i=i+1`, срабатывает инструкция `i=5`, и значение переменной `i` снова становится равным 5.



- ✓ Вот как, вероятно, инструкцию `for` хотел написать программист:

```
for (i=1; i<=5; i=i+1)
```

Эта строка повторяет цикл пять раз.

- ✓ Некоторые компиляторы иногда могут обнаружить условие бесконечного цикла в инструкции `for` и пометить такую инструкцию как бесконечный цикл. Если ваш компилятор это может, вам сильно повезло. Например, старый компилятор Borland C++ отметил в `FOREVER.C` возможное наличие бесконечного цикла так: `Possibly incorrect assignment error` (Возможная ошибка: неправильное присваивание). Однако компилятор все равно генерирует объектную (неправильную, конечно) программу.

Выход из цикла

Цикл не бесконечен, если есть способ прервать (`break`) его. Для большинства циклов условие выхода определяется в самой инструкции цикла. В инструкции `for` условие выхода определяется средним элементом цикла. Вот пример:

```
for (a=32; a<128; a=a+1)
```

Условие выхода в этом цикле определяется условием `a<128`; условие выхода из цикла — это условие, при выполнении которого цикл заканчивается.

Некоторые циклы, однако, разработаны так, что не имеют конца. Дело в том, что условие, при котором заканчивается цикл, вычисляется где-нибудь в другом месте. В этом случае цикл разрабатывается так, чтобы он выполнялся вечно, — и выглядит это прекрасно, если только некоторое условие в другом месте в конечном счете не прерывает цикл.

Приведу пример. Большинство текстовых процессоров представляют собой программы со скрытыми бесконечными циклами. Цикл много раз проверяет ввод с клавиатуры — он ждет, пока вы напечатаете какую-нибудь команду. Только когда вы напечатаете команду выхода из текстового процессора, эта команда остановит текстовый процессор, и вы возвратитесь в DOS. Это похоже на управляемый бесконечный цикл — на самом деле, конечно, не бесконечный, потому что из него есть выход.

Следующая программа — `TYPER1.C`, примитивный текстовый редактор. (Настолько примитивный, что он хоть и напоминает текстовый процессор, но не более чем кукла может напоминать живого человека.) В этом текстовом редакторе возможен лишь набор текста и его отображение на экране, и ничего более. В самой программе нарочно используется бесконечный цикл `for`. Чтобы выйти из цикла, когда пользователь нажимает клавишу `<->` (тильда), используется ключевое слово `break` в команде `if`:

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

```
int main()                                // главная функция
{
    char ch;

    puts("Start typing");                  // Запустить печать
    // Нажать <->, затем <Enter>, чтобы остановиться
    puts("Press ~ then Enter to stop");
    for(;;)
    {
        ch=getchar();
        if(ch=='~')                        // если (ch == '~')
        {
            break;
        }
    }
}
```

```

}
printf("Thanks!\n");           // Спасибо!
return(0);
}

```

Среди программ этой книги данная программа является чемпионом по количеству фигурных скобок и отступов. Будьте внимательны при вводе данной программы с помощью редактора. Сохраните файл на диске под именем `TYPER1.C`.

Скомпилируйте `TYPER1.C`.

Выполните полученную программу. Вы увидите, что данная программа работает подобно пишущей машинке. Вы можете напечатать довольно много символов и заполнить весь экран текстом.

Когда вы закончите печать, нажмите клавишу с тильдой (~), а затем — <Enter>. (Это единственная команда вашего “печатающего устройства” — программы `TYPER`.) Теперь вы действительно закончили.

- ✓ Инструкция `for(;;)` не является ошибочной. Дело в том, что точки с запятой обязательны, но все, что стоит между ними, можно опустить (это особенность ключевого слова `for`).
- ✓ Вслух я читаю команду `for(;;)` “for ever” (“вечно”).
- ✓ Цикл `for` в `TYPER1.C` бесконечен: его условие отсутствует, а потому рассматривается как (вечно) истинное, поскольку проверять нечего. Отсутствие условия в цикле `for` ведь не является синтаксической ошибкой! Соответственно, компилятор предполагает, что условие истинно все время (т.е. всегда принимает значение `true` (истина)), и поэтому такой цикл в программе повторяется бесконечно.
- ✓ Первая и последняя части (элементы инструкции `for`) также отсутствуют, хотя точки с запятой в круглых скобках должны быть обязательно — таковы требования этикета в языке программирования C.
- ✓ Поскольку циклу `for` принадлежат несколько инструкций, они заключены в фигурные скобки.
- ✓ Функция `getchar` ждет нажатия клавиши на клавиатуре и отображает на экране символ, соответствующий нажатой клавише. Затем символ сохраняется в переменной `ch`.
- ✓ Условный оператор проверяет, равно ли значение переменной `ch` символу ~ (тильда). Обратите внимание, что используются два знака “=”. Если в результате операции сравнения будет получено значение `true` (истина), то это укажет на то, что пользователь нажал клавишу ~, и будет выполнена команда `break`. В противном случае `break` пропускается, и цикл повторяется — с клавиатуры считывается еще один символ.

Ключевое слово `break`

Ключевое слово `break` языка C позволяет выйти из любого цикла, а не только из цикла `for`. Независимо от условия окончания цикла, `break` немедленно запускает команду “выйти из цикла”. Программа продолжает выполняться со следующей инструкции после цикла:

```

break;                               // прервать

```

Ключевое слово `break` само по себе является инструкцией языка C и потому после него обязательно должна стоять точка с запятой.

Ключевым словом `break` можно прервать не только бесконечный цикл, но и любой другой цикл. Иными словами, с помощью этого ключевого слова можно выйти из любого цикла, а не только избежать заикливания. Инструкция `break` может остановить любой цикл в любое время, поэтому она часто используется в условном операторе, который проверяет некоторое условие. В зависимости от результата проверки цикл может быть остановлен с помощью `break`. (Именно так было сделано в программе `TYPERS.C` в предыдущем разделе.)



- ✓ Инструкция `break` останавливает только тот цикл, в котором она находится. Она не позволяет выйти из вложенного цикла, или цикла внутри другого цикла. Более подробно вложенные циклы рассматриваются в главе 18 “Циклы с условием продолжения. Организация задержки”.
- ✓ Если ключевое слово `break` встретится вне цикла, т.е. там, где нет ничего такого, из чего можно выйти, компилятор обидится и сгенерирует сообщение об ошибке.
- ✓ Забавно, что используется английское слово `break`, а не *brake* (тормоз, задерживать). Такая же несколько неожиданная логика, вероятно, применялась и тогда, когда давали название клавише прерывания на клавиатуре (клавише `<Break>`).

Знакомство с циклами и применением инкремента (оператора ++) в циклах

В этой главе...

- Приращение переменных с помощью оператора ++
- Уменьшение переменных с помощью оператора --
- Использование других сокращений для других математических действий

П

ак же, как поиск компромисса — ключ к продвижению в политике, выполнение цикла — основная часть программирования. И мы уже подошли близко к выполнению цикла — вы уже познакомились со всеми необходимыми понятиями, главным из которых является приращение, означающее увеличение чего-либо.

Когда в цикле выполняются различные итерации, переменные увеличиваются или уменьшаются, чтобы компьютер мог следить за ходом выполнения циклов. Фактически, концепции выполнения цикла и приращения связаны настолько близко, что нелегко было писать предыдущую главу о команде `for`, пытаясь избежать упоминания циклов. Теперь пришло время познакомить вас с древним искусством и таинственной практикой приращения.

Искусство приращения

Когда цикл `for` повторяет что-либо семь раз, какая-нибудь переменная увеличивается семь раз. Например:

```
for(i=0;i<7;i=i+1)
```

Эта инструкция `for` — заголовок цикла, который повторяется семь раз, от `i=0` с приращением 1 семь раз, пока значение `i` меньше, чем 7 (`i<7`).

Если вы находите странным начинать отсчет циклов с 0, следующая инструкция `for` выполняет тот же самый цикл семь раз:

```
for(i=1;i<=7;i=i+1)
```

В этом примере `i` увеличивается от 1 до 7. Знатки языка C предпочитают начинать отсчет переменной цикла с 0, потому что так считает сам компьютер (он любит начинать счет с 0). Но как бы то ни было, приращение — центральная идея выполнения цикла.



Имейте в виду, что инструкция `for` — просто рамка для цикла. Она повторяет группу инструкций заданное число раз. Инструкция `for` сама по себе только управляет выполнением цикла.

Загадочные символы операторов языка С, том I: оператор приращения (++)

Искусство приращения состоит в том, чтобы добавить 1 к значению переменной. Независимо от первоначального значения переменной count (счет), после выполнения следующей инструкции оно будет на 1 больше:

```
count=count+1;
```

Взгляните на эту инструкцию: на это равенство противно даже смотреть. И все же ни один цикл не может обойтись без него, а отсюда следует, что приращение часто встречается в программах на С. Но несмотря на это, лишь немногие программисты в С используют предыдущую инструкцию. Вместо нее они прибегают к сокращенной записи оператора приращения — двум знакам “плюс”, взявшимся за руки:

```
count++;
```

```
// счет++
```

Оператор приращения, ++, работает подобно другим математическим операторам, которые вы, возможно, видели в других неприятных математических главах: +, -, * и / для обозначения соответственно сложения, вычитания, умножения и деления. Различие здесь в том, что ++ работает без знака “=”. Он только говорит компилятору: “Эй! Добавь 1 к значению этой переменной. Спасибо. Большое спасибо.” Это быстро и опрятно, но немного загадочно (именно поэтому я не познакомил вас с ним сразу же).

- ✓ Да, вы можете читать “плюс плюс”, когда видите ++ в программе.
- ✓ Да, именно поэтому С++ называют “С плюс плюс.” В этом также изюминка шутки: С++ — это “на один больше”, чем обычный С.
- ✓ Если вы используете ++, знак “=” не нужен. Просто поставьте ++ после переменной, и эта переменная будет увеличена на 1.
- ✓ В виде равенства i++ записывается как i=i+1.
- ✓ Вот что мы имеем:

```
var=3; /* переменная var равняется трем */
var++; /* Ой! Переменная var увеличена здесь */
      /* Теперь переменная var равняется четырем */
```

- ✓ Оператор ++ в инструкции for используется так:

```
for (i=0; i<7; i++)
```

Эта строка означает, что переменная i увеличивается в каждой итерации цикла.
- ✓ Мы вступаем в область, где язык С начинает становиться действительно загадочным. Читая отдельные части цикла for, даже самые образованные люди могут подумать, что i=1 значит “i равно 1” и что i<7 значит “i меньше, чем 7” и даже что i=i+1 значит “i равно i плюс 1.” Но подбросьте им i++ и они подумают: “i плюс плюс? Сверхъестественно.”



Еще один просмотр программы LARDO.C

В главе 11 “Больше математики и Священный порядок (старшинство операций)” мы уже встречались с идеей увеличения переменной в программе LARDO.C. Я уверен, что эта программа близка и дорога вашему сердцу и неоднократно впечатляла множество ваших друзей и, конечно же, все ваше семейство. К сожалению, теперь, когда вы знакомы с оператором ++, вы понимаете, что такая программа действительно могла бы смутить знатоков С, поскольку

все те неловкие `w=w+1` должны быть заменены командами `w++`. Коротко. Конфетка. Загадочно. В конце концов, это ведь именно так компьютеры представляются всем!

Следующая программа — обновленный исходный текст `LARDO.C`, который, вероятно, все еще где-нибудь валяется на вашем жестком диске. Загрузите тот старый файл с помощью редактора и сделайте необходимые изменения, чтобы программа имела следующий исходный текст:

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

int main()                                // главная функция
{
    char weight[4];
    int w;

    printf("Enter your weight:");          // Введите ваш вес
    gets(weight);                          // вес
    w=atoi(weight);
    // Ваш вес сейчас
    printf("Here is what you weigh now: %i\n",w);
    w++;
    // Ваш вес после картофеля
    printf("Your weight after the potatoes: %i\n",w);
    w++;
    // Вот, пожалуйста, после баранины
    printf("Here you are after the mutton: %i\n",w);
    w=w+8;
    // И ваш вес после десерта
    printf("And your weight after dessert: %i pounds!\n",w);
    printf("Lardo!\n");
    return(0);
}
```

Отредактируйте исходный текст. Изменение состоит в замене двух строк.

Сохраните файл на диске со старым названием (именем), потому что эта программа намного лучше первоначальной. Затем скомпилируйте.

Исправьте все ошибки, если они есть. Конечно, программа выполняется так же, как и прежде. Единственное истинное различие состоит в том, что теперь используются преимущества оператора приращения `++`, можете показывать вашу программу программистам во всем мире и хвастаться своим знанием языка C.



Обратите внимание, что инструкция `w=w+8` не изменилась. Причина состоит в том, что переменная `w` увеличивается на 8, а не на 1. Да, для этой операции тоже есть сокращение, но я не покажу его вам до конца этой главы.

Главная практика уменьшения

Циклы не обязательно должны продвигаться вперед. Они также могут считать в обратном порядке, который иногда является определенно более естественным и требуется в некоторых ситуациях (например, запуск космического корабля, а также другая работа, которую вы делаете каждый день).

Рассмотрим программу `OLLYOLLY.C`, которая ведет отсчет в обратном порядке. Собственно говоря, она больше ничего и не делает:

```
#include <stdio.h>
int main()                                // главная функция
```

```
{
    int count;                                // счет

    for(count=10; count>0; count=count-1)
        printf("%d\n", count);                // счет

    // Готовы или нет, сейчас я прибываю!
    printf("Ready or not, here I come!\n");
    return(0);
}
```

Создайте в редакторе новый документ и тщательно напечатайте этот исходный текст. Здесь все для вас знакомо, кроме, может быть, того, что находится в круглых скобках цикла `for`, которые могут выглядеть немного необычно — но это и есть обратный отсчет! Цикл будет работать? Компьютер взорвется? Джейн действительно обманывает Ральфа? Как это может случиться?

Выйдите из обдумывания и печатайте.

Сохраните файл на диске под именем `OLLYOLLY.C`. Скомпилируйте и выполните его.

Вывод выглядит так:

```
10
9
8
7
6
5
4
3
2
1
Ready or not, here I come!
```

(Готовы или нет, сейчас я прибываю!)

Да, действительно, компьютер может считать назад. И делает он это в цикле `for`, притом с минимумом суеты — но все же с некоторым раздражением. Подробности обсуждаются в следующем разделе.

- ✓ Чтобы доказать, что вы не сходите с ума, обратитесь к программе `100.C` из главы 15 “Циклы в языке программирования C”. Она считает от 1 до 100 в цикле `for`. Единственное отличие, если не считать инструкции `printf()`, состоит в том, что цикл ведет отсчет в обратном направлении.
- ✓ Циклы, отсчитывающие в обратном направлении, в программах на C встречаются редко. В большинстве случаев всякий раз, когда вы должны сделать кое-что 10 раз, вы делаете это, считая от 0 до 9 или от 1 до 10 или как-то иначе, однако обычно в цикле `for` вы ведете отсчет в порядке возрастания.

Отсчет в обратном порядке

Для компьютера не имеет значения, считать назад или вперед. Вы только должны указать на языке C, в каком направлении вы хотите вести отсчет.

Чтобы считать в прямом направлении, вы увеличиваете значение переменной. Так, если вы имеете переменную `f`, вы делаете это следующим образом:

```
f=f+1;
```

или даже вот так:

```
f++;
```

Как бы то ни было, значение переменной `f` должно быть на 1 больше прежнего; это и есть увеличение переменной.

Чтобы считать назад, вы вычитаете 1 из значения переменной: 10, 9, 8, 7 и т.д. Инструкция выглядит идентично инструкции приращения, за исключением знака “минус”:

`b=b-1;`

Значение переменной `b` на 1 меньше прежнего. Если `b` имело значение 5, инструкция устанавливает значение `b` равным 4. Этот процесс называется уменьшением значения переменной.

- ✓ Уменьшение, или вычитание 1 (или любого другого числа) из значения переменной представляет собой самое обычное вычитание. Единственное отличие состоит в том, что уменьшение выполняется в цикле, и поэтому цикл ведет отсчет в обратном направлении.
- ✓ Увеличить — это значит добавить (1) к значению переменной.
- ✓ Уменьшить означает вычесть (1) из значения переменной.
- ✓ Уменьшение работает, потому что C сначала вычисляет то, что находится справа от знака “=”:

`b=b-1;`

Сначала вычисляется `b-1`, так что компьютер вычитает 1 из значения переменной `b`. Затем это значение “перепрыгивает” через знак “=” назад, в переменную `b`. Таким образом переменная уменьшается.

Цикл: отсчет в обратном порядке

Программа `OLLYOLLY.C` содержит пример отсчета в обратном порядке в цикле `for`:

```
for (count=10; count>0; count=count-1)
```

Это типичный пример цикла `for`. Цикл состоит из следующих частей: инициализации, условия и приращения. Части цикла приведены в табл. 16.1.

Таблица 16.1. Отсчет в обратном порядке в цикле `for`

Часть заголовка цикла	Инструкция
Инициализация	<code>count=10</code>
Условие	<code>count>0</code>
Приращение	<code>count=count-1</code>

В цикле при отсчете в обратном порядке все работает так, как указано в заголовке. В начале цикла значение переменной `count` (счет) устанавливается равным 10. Выполнение цикла продолжается, пока значение переменной `count` больше 0 (условие `count>0`). Однако в конце каждого повторения цикла значение переменной `count` уменьшается. Так ведется отсчет в обратном порядке.

И снова нет особых причин в цикле вести отсчет в обратном порядке — за исключением того, что инструкция `printf()`, принадлежащая циклу, отображает числа от 10 до 1 путем отсчета в обратном направлении. Обычно (что означает приблизительно 99 процентов случаев) в цикле отсчет ведется вперед. Это не только проще для человека, но и позволяет уменьшить вероятность ляпа при программировании, которая возрастает при использовании отсчета в обратном порядке из-за непривычки.



- ✓ В большинстве циклов счет ведется в порядке возрастания. Цикл в С с отсчетом в обратном порядке допускается, но используется редко, главным образом потому, что счет есть счет, и для людей считать более просто в порядке возрастания.
- ✓ Чтобы в цикле вести отсчет в обратном порядке, необходимо уменьшать переменную цикла.
- ✓ Цикл с отсчетом в обратном направлении отличается не только тем, что уменьшается переменная цикла, но и условием цикла (средняя часть заголовка цикла). (Других отличий нет.) Цикл с отсчетом в обратном направлении требует немного больше умственных усилий, чем обычный цикл `for`. Это еще одна причина, по которой этот тип цикла встречается редко.
- ✓ Хорошо. Возникает вопрос: “Зачем же беспокоиться?” Просто потому, что вы должны быть знакомы с уменьшением и загадочным оператором `--`, который описан в следующем разделе.

Загадочные символы операторов языка С, том II: оператор уменьшения (`--`)

Точно так же, как в С есть сокращение для увеличения значения переменной, есть и сокращение для уменьшения значения переменной. Чтобы добавить 1 к значению переменной — *увеличить* его, используется следующая инструкция языка С:

```
i=i+1;
```

Если вы помните сокращение, вы можете использовать загадочный оператор `++`, чтобы сделать то же самое:

```
i++;
```

В обоих примерах к значению переменной `i` добавляется 1.

Рассмотрим следующий пример:

```
d=d-1
```

Эта инструкция вычитает 1 из значения переменной `d`, уменьшая его. Сокращенно это записывается так:

```
d--;
```

Оператор декремента `--` вычитает 1 из значения переменной.

Точно так же как оператор приращения `++`, оператор декремента `--` работает подобно другим математическим операторам в языке С. Чтобы все было загадочным, знак “=” не ставится. Оператор уменьшения `--` указывает, что нужно вычесть 1 из значения связанной с ним переменной, — это будет сделано даже без явного указания знака “=”.

- ✓ Символы `--` можно читать “минус-минус”, или “уменьшить”, или “декремент”, или даже “дек дек”, хотя специального соглашения в мире программистов по этому вопросу нет. Некоторые считают неприличным говорить “минус-минус” — во всяком случае, вслух.
- ✓ Выражение `d--` — это то же самое, что и `d=d-1`.
- ✓ Штраф за использование `d=d-1` вместо `--` законом не предусмотрен.
- ✓ Если используете оператор уменьшения `--`, не ставьте знак “=”. Просто после переменной, которую вы хотите декрементировать, поставьте `--` и все будет сделано автоматически!

✓ Вот что мы имеем:

```
var=3; /* значение переменной var равняется трем */  
var--; /* Стоп! Переменная var здесь уменьшается */  
      /* Теперь значение переменной var равняется двум */
```

✓ Оператор `--` в инструкции `for` используется так:

```
for (d=7; d>0; d--)
```

Выражение `d--` заменяет `d=d-1`, как уже было продемонстрировано в нескольких предыдущих разделах.

Заключительное усовершенствование OLLYOLLY.C

Теперь, когда вы знакомы с `--`, вы можете заменить неуклюжую и несколько смущающую программу OLLYOLLY.C первоклассной программой с оператором уменьшения.

Загрузите OLLYOLLY.C с помощью вашего редактора (если сейчас она еще не в окне редактора), найдите `count=count-1` (это в самом начале цикла `for`). Отредактируйте инструкцию `for`, заменяя `count=count-1` загадочным, хотя правильным, выражением `count--`. После редактирования строка должна выглядеть так:

```
for (count=10; count>0; count--)
```

Сохраните файл исходного текста на диске, а затем перетранслируйте его.

Программа выполняет то же самое, но ваши приятели, программирующие на C, будут изумлены вашей ловкостью в использовании оператора уменьшения.

Больше безумия приращения

Приращение и выполнение цикла связаны так же крепко, как голливудские актеры, избражающие самую горячую влюбленную парочку.

Выполнение цикла — важная часть программирования. Выполнение инструкций напоминает репетицию, во время которой известный актер заучивает свою роль. В C все это можно сделать с помощью цикла `for`, увеличивая (или уменьшая — декрементируя) значение переменной.

При выполнении цикла работают операторы `++` и `--`, которые, впрочем, можно также использовать и независимо от выполнения цикла — для того, чтобы увеличить или уменьшить значение переменной (так, как некоторые актрисы увеличивают размер своего бюста при помощи различных хирургических методов и уменьшают возраст с помощью грима).

Учитывая это, вы должны иметь в виду, что приращение и уменьшение не всегда изменяет значение переменной непременно на 1. Например, в программе LARDO.C значение переменной `w` увеличивается на 8, для этого используется следующая инструкция:

```
w=w+8;
```

Значение переменной `w` увеличено на 8. Следующая инструкция уменьшает `w` на 3:

```
w=w-3;
```

Это все равно приращение и уменьшение, хотя значения отличаются большие, чем на 1. Эти операторы совершенно правильны. И эти типы приращения и уменьшения можно использовать в циклах. И сейчас (немедленно!) будет приведен пример.

✓ Хотя значение переменной можно увеличить более чем на 1, оператор `++` увеличивает значение переменной только на 1. То же самое верно для оператора `--`, который всегда уменьшает значение переменной на 1.

✓ К счастью, в языке C нет операторов `+++` или `---`. Забудьте о них!

Прыгающие циклы!

Вот арифметическая прогрессия: 2, 4, 6, 8.

Следующая программа использует цикл `for`, чтобы генерировать арифметическую прогрессию 2, 4, 6, 8. Именно в цикле `for` переменная перескакивает через одно число. Именно такой цикл я называю прыгающим или подскакивающим:

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

```
int main() // главная функция
{
    int i;

    for(i=2;i<10;i=i+2)
        printf("%d ",i);
    // Кто выиграл?
    printf("who do we appreciate? GNU!\n");
    return(0);
}
```

Выберите New (создать) в вашем редакторе и введите предыдущий исходный текст. В инструкции `printf()` внутри цикла `for` обратите внимание на пробел после `%d`. Там идут следующие знаки: “двойная кавычка, знак процента, строчная (малая) буква `d`, пробел, двойная кавычка”. Сохраните файл на диске под именем `CHANT.C`.

Скомпилируйте и выполните программу. Вот как будет выглядеть вывод:

```
2 4 6 8 who do we appreciate? GNU!
```

```
(2 4 6 8 кто выиграл? ГНУ!)
```

- ✓ Цикл начинается с 2, приращения выполняются до 10 с помощью инструкции `i=i+2`. Цикл читается так: “Начать с `i` равного 2, и пока значение `i` меньше 10, повторять следующие инструкции, добавляя 2 к переменной `i` каждый раз в конце цикла”.
- ✓ Вы можете изменить заголовок цикла в программе так, чтобы программа считала четные числа в цикле до любого значения. Например:

```
for(i=2;i<1000;i=i+2)
```
- ✓ Эта модификация заставляет компьютер считать двойками от 2 до 998.

Считаем до 1 000 пятерками

Следующая программа — обновление старой программы `100.C` из главы 15 “Циклы в языке программирования C”. В данной версии программа считает до 1 000 пятерками — без помощи компьютера на выполнение этой задачи ушло бы несколько дней:

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

```
int main() // главная функция
{
    int i;

    for(i=5;i<=1000;i=i+5)
        printf("%d\t",i);
    return(0);
}
```

Начните с нового, чистого экрана в вашем редакторе. Напечатайте предыдущий исходный текст. В нем нет ничего, что потрясло бы ваше воображение. Действительно, это ведь только новый вариант старой программы 100.C. Сохраните файл на диске под именем 1000.C.

Скомпилируйте 1000.C и выполните полученную программу. Ваш экран заполняется значениями от 5 до 1000, все они выстраиваются в колонну (по строкам и столбцам).

- ✓ Этот прыгающий (подскакивающий) цикл считает пятерками, потому что приращение цикла `for` определяет инструкция `i=i+5`. Операция `i=i+5` увеличивает значение переменной `i` на 5.
- ✓ Цикл начинает счет с 5, поскольку инициализацию цикла `for` определяет инструкция `i=5`. Он прекращает счет до 1 000 из-за условия цикла `i<=1000`. Это значит “меньше или равно 1000”. Именно это условие позволяет досчитать до 1 000.

Загадочные символы операторов языка C, том III: безумие продолжается

Язык C полон сокращений, особенно часто они используются для математических операций. Конечно, два самых загадочных сокращения предназначены для изменения значения переменной на 1: `++` — для увеличения его на 1, и `--` для его декремента. Но их намного больше!

В программе 1000.C для добавления 5 к значению переменной, например, используется следующая инструкция:

```
i=i+5
```

Загадочное сокращение в языке C для этой операции:

```
i+=5
```

Эта строка означает: “Увеличить значение переменной `i` на пять.” К сожалению, это даже близко не похоже на то, что оно означает.

Хотя я могу написать `++`, чтобы увеличить переменную, и `--`, чтобы декрементировать ее, сочетание вроде `+=` очень напоминает опечатку. Грустные новости: это не опечатка. Есть даже более грустные новости: аналогичные операторы предусмотрены для добавления к значению переменной некоторого числа (или другой переменной), вычитания из значения переменной некоторого числа (или другой переменной), умножения значения переменной на некоторое число (или другую переменную) и деления значения переменной на некоторое число (или другую переменную). В табл. 16.2 приведен целый список таких сокращений.

Таблица 16.2. Загадочные сокращения для обычных математических операций

<i>Длинная, скучная инструкция</i>	<i>Загадочное сокращение</i>
<code>var=var+5</code>	<code>var+=5</code>
<code>x=x+y</code>	<code>x+=y</code>
<code>var=var-5</code>	<code>var-=5</code>
<code>x=x-y</code>	<code>x-=y</code>
<code>var=var*5</code>	<code>var*=5</code>
<code>x=x*y</code>	<code>x*=y</code>
<code>var=var/5</code>	<code>var/=5</code>
<code>x=x/y</code>	<code>x/=y</code>

В табл. 16.2 приведено по два примера для каждого загадочного сокращения. В первом примере переменная `var` изменяется с помощью постоянного значения 5. Во втором примере используется две переменные; первая `x` изменяется с помощью значения другой переменной `y`.

Да, сокращения для приращения, уменьшения и изменения переменной являются загадочными. Вы не обязаны использовать их. Вы не будете оштрафованы, если забудете о них. Я рассказываю о них здесь по двум причинам: вы можете освоить их, а знатоки языка C любят украшать ими свои программы; так что не пугайтесь, когда увидите их.

Самостоятельная работа: измените предыдущие две программы, `CHANT.C` и `1000.C`. Замените длинное математическое выражение в цикле `for` его сокращенной версией. Ответы найдете в конце этой главы.



- ✓ В технических руководствах все эти штучки (сокращения) называются операторами присваивания. Не запоминайте этот термин. Даже я должен был искать его.
- ✓ Не забудьте с помощью липкой ленты прикрепить табл. 16.2 к стене. Сделайте также закладку в книге на соответствующей странице. Эти загадочные сокращения не просто запомнить.
- ✓ Чтобы запомнить, что сначала идет арифметический оператор (+, -, * или /), посмотрите на неправильный способ вычитания:

```
var=-5
```

Это совсем не сокращение для `var=var-5`. Вместо вычитания эта инструкция устанавливает значение переменной `var` равным минус пять. Правильно писать так: `var-=5`.

- ✓ Помните, что эти математически-сокращенные загадочные операторы можно использовать не только в цикле `for`. Каждый из них может быть инструкцией языка C сам по себе, или математической операцией, предназначенной для изменения значения переменной. Вот так:

```
term+=4;
```

Эта инструкция увеличивает значение переменной `term` на 4.

Ответы

В `CHANT.C` измените заголовок цикла так:

```
for(i=2;i<10;i+=2)
```

В `1000.C` измените заголовок цикла так:

```
for(i=2;i<10;i+=5)
```

В обоих случаях вы заменяете более длинное выражение `i=i+x` его более короткой записью `i+=x`.

Познакомьтесь с циклом `while` (циклом с условием продолжения)

В этой главе...

- Использование цикла с условием продолжения
- Выбор между циклом `for` и циклом с условием продолжения
- Создание бесконечных циклов с условием продолжения
- Пока (за работу платят вперед)...

В языке C предусмотрено несколько типов циклов. Для вашей программы вы можете выбрать сложный цикл `for`, который имеет все необходимое для того, чтобы сделать большинство программистов счастливыми, или же вы можете выбрать более экзотический, вольный цикл с условием продолжения. Примите во внимание, что цикл `for` более официальный и содержит все свои параметры в одном месте, в то время как цикл `while` (цикл с условием продолжения) причудлив и свободен — он подобен тем беззаботным юнцам, все проделки которых поощряются мамой и оплачиваются папой.

В этой главе вы познакомитесь с беспечным циклом с условием продолжения. Выполнение цикла — понятие (концепция), с которым вы должны быть уже знакомы, если вы потрудились над циклами `for` в нескольких предыдущих главах. У меня для вас хорошие новости! Циклы с условием продолжения намного проще! Ха-ха! Итак...

Цикл `while` — цикл с условием продолжения

Циклы с условием продолжения (циклы `while`) не должны пугать вас. Рассмотрим следующий пример:

Пока (`while`) горит красный, держите ногу на тормозе.

Это простой пример (из реальной жизни) цикла с условием продолжения, он означает, что “пока условие выполнено (истинно), продолжайте повторять данное действие” (то есть цикл). Вы держите ногу на тормозе, пока горит красный, — это и есть цикл. Достаточно просто.

На языке C несложно переписать эту команду, используя цикл с условием продолжения. Это может выглядеть следующим образом:

```
while(light==RED)                // пока (свет == КРАСНЫЙ)
{
    foot_on_brake();              // держите ногу на тормозе
    light = check_light();        // свет
}
```

Конструкция `light==RED` (свет == КРАСНЫЙ) — условие, которое может быть либо истинным (принимать значение `TRUE` (ИСТИНА)), либо ложным (принимать значение `FALSE` (ЛОЖЬ)) — как в условном операторе. Пока это условие истинно, повторяется выполнение инструкций внутри фигурных скобок цикла с условием продолжения.

Одна из инструкций в цикле с условием продолжения изменяет условие повторения цикла: `light=check_light()` модифицирует состояние переменной `light` (свет). Когда ее значение изменится, т.е. станет чем-нибудь отличным от `RED` (КРАСНЫЙ), условие в инструкции `while` станет ложным, блок инструкций будет пропущен, зато будет выполнена следующая часть программы. Это и есть суть цикла с условием продолжения в `C`.

Выполняя цикл `while` часами

Как и цикл `for`, цикл с условием продолжения можно настроить так, чтобы повторить блок инструкций данное число раз. В отличие от цикла `for`, рычаги управления циклом с условием продолжения (это выражение, которое указывает циклу, когда он должен начаться и когда закончиться) могут быть размещены повсюду. С одной стороны, это хорошо, потому что их не обязательно все втискивать в одну строку, как для цикла `for`, с другой — плохо (об этом вы узнаете в следующем разделе). Пока же займитесь вводом исходного текста `HEY.C`, блестящей программой на `C`:

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

```
int main()                                // главная функция
{
    int i;

    i=1;
    while(i<6)                             // пока (i<6)
    {
        // Ой! Пожалуйста, остановитесь!
        printf("Ouch! Please stop!\n");
        i++;
    }
    return(0);
}
```

Напечатайте эту программу, которая по существу является обновлением программы `OUCH.C` из главы 15 “Циклы в языке `C`”. Фактически обе программы делают то же самое, но в них используются различные типы циклов. Сохраните файл на диске под именем `HEY.C`.

Скомпилируйте и выполните.

Вот что получится:

```
Ouch! Please stop!
Ouch! Please stop!
Ouch! Please stop!
Ouch! Please stop!
Ouch! Please stop!
```

```
(Ой! Пожалуйста, остановитесь!
Ой! Пожалуйста, остановитесь!
Ой! Пожалуйста, остановитесь!
Ой! Пожалуйста, остановитесь!
Ой! Пожалуйста, остановитесь!)
```

Бриллиант. Просто бриллиант.



- ✓ Цикл на языке С состоит из трех частей: начала, средней части (часть, которая повторяется) и конца. Это вы помните из главы 15 “Циклы в языке С”.
- ✓ Информация, управляющая выполнением цикла `for`, находится прямо в круглых скобках команды `for`.
- ✓ Информация, управляющая выполнением цикла с условием продолжения, находится перед циклом, в круглых скобках команды `while`, и внутри самого цикла с условием продолжения.
- ✓ В НЕУ.С переменная `i` установлена (инициализирована) перед циклом. Затем она увеличивается в цикле. Сама инструкция `while` выполняется только в том случае, если `i` меньше 6.
- ✓ Инструкции, принадлежащие циклу `while`, — это те строки, которые находятся внутри фигурных скобок — повторяются, пока условие внутри круглых скобок истинно. Если условие ложно (например, если переменная `i` была уже больше 6), то эти инструкции пропускаются (как в условном операторе).
- ✓ Циклы с условием продолжения могут запросто стать причиной ляпа, называемого бесконечным циклом. По этой причине вы должны удостовериться в том, что условие, проверяемое циклом `while`, в конечном счете станет ложным, и это заставит цикл прекратить повторение.
- ✓ Чтобы выйти из цикла с условием продолжения, в цикле должно случиться нечто такое, что изменит значение условия, проверяемого циклом `while`.
- ✓ Вероятно, вы впервые встретили две различных программы, которые решают ту же самую проблему: в ОУС.С используется `for`, а в НЕУ.С используется цикл с условием продолжения. Ни одна из этих программ не лучше, чем другая. Дополнительная информация содержится в этой же главе в разделе “Выбор типа цикла: цикл с условием продолжения или цикл `for`”.



Ключевое слово `while` (формальное введение)

В языке С ключевое слово `while` используется, чтобы повторить блок инструкций. В отличие от цикла `for`, `while` указывает компьютеру только когда нужно закончить цикл. Цикл должен быть настроен перед ключевым словом `while`, а во время выполнения цикла должно проверяться условие окончания цикла — в этом оно похоже на шипящий плавкий предохранитель или тикающий таймер. Затем продолжается выполнение цикла до того момента, когда условие, проверяемое ключевым словом `while`, внезапно не станет ложным. Тогда цикл заканчивается, и программа продолжает выполняться (несколько более грустно, но все же довольная тем фактом, что у нее была возможность повторяться некоторое время).

Ниже приведен примерный формат:

```
старт;  
while(условие)                                // пока (условие)  
{  
    инструкция (и);  
    управление;  
}
```

Сначала цикл должен быть настроен, это делается в инструкции *старт*. Например, эта инструкция (или группа инструкций) может объявить, что переменная будет иметь некоторое значение, что она будет ждать нажатия клавиши или сделает много других интересных вещей. *Условие* — проверяемое условие. Если условие истинно, повторяются *инструкции*, заключенные в фигурные скобки. После того как эти инструкции повторены (завершен очеред-

ной цикл), ключевое слово `while` проверяет указанное условие, причем останов цикла происходит только тогда, когда инструкция `while` обнаружит, что условие ложно.

В фигурных скобках — инструкции, повторяемые циклом с условием продолжения. Одна из инструкций, *управление*, управляет циклом. Так или иначе, *управление* должно изменить *условие*, точнее его значение, чтобы цикл, в конечном счете, остановился (произошел выход из цикла).

Циклы с условием продолжения имеют преимущество перед циклами `for`, поскольку их легче читать по-английски. Например:

```
while(ch!='-')
```

Эта инструкция велит: “пока значение переменной `ch` не равняется символу тильды, повторяйте следующие инструкции”. Чтобы понять смысл инструкции, вы должны помнить, конечно, что в С этот `!` означает *не*. Для понимания программ на С важно знать, какие символы произносятся, а какие являются только художественными украшениями.

- ✓ Истинность условия цикла с условием продолжения проверяет так же, как это делает и ключевое слово `if`. В частности, в условии можно написать неравенство. Символы, используемые в условном операторе, используются и в цикле с условием продолжения. В частности, можно использовать `==`, `<`, `>` и `!=`. Вы можете даже использовать логические операторы `&&` (знак логической операции AND (И)) или `||` (знак логической операции OR (ИЛИ)) для вычисления логических выражений, составленных из нескольких неравенств.

- ✓ Вот какой формат цикла с условием продолжения в HEY.C:

Старт: `i=1`

Условие: `i<6`

Управление: `i++`

- ✓ Обратите внимание, что, подобно ключевому слову `for`, после `while` точка с запятой не ставится. После него сразу следует группа инструкций в фигурных скобках.

- ✓ Если цикл с условием продолжения содержит только одну инструкцию, вы можете обойтись без фигурных скобок:

```
старт;  
while(условие)                                // пока (условие)  
    управление;
```

- ✓ Некоторые циклы с условием продолжения вообще не имеют никаких инструкций. Это довольно обычное явление:

```
while((условие)==TRUE)                        // пока ((условие)==TRUE)  
    ;
```

В этом примере точка с запятой — “инструкция”, которая принадлежит циклу с условием продолжения.



Выбор типа цикла: цикл с условием продолжения или цикл `for`

Цикл с условием продолжения и цикл `for` во многом подобны. Определенно нужно свихнуться, чтобы использовать цикл с условием продолжения, если вы любите цикл `for`, и наоборот. Вот пример цикла `for` из программы 100.C (глава 15 “Циклы в языке С”):

```
for(i=1;i<=100;i=i+1)  
    printf("%d\t",i);
```

Этот цикл считает от 1 до 100 и отображает каждое число.

А вот тот же самый цикл, но в формате while:

```
i=1;
while(i<=100)                                // пока (i<=100)
{
    printf("%d\t",i); i=i+1;
}
```

Обратили внимание, как цикл for был разбит на части и помещен в цикл с условием продолжения? Чтобы вам было проще увидеть это, я обозначил части прописными (большими) полужирными буквами. Вот что у меня получилось:

```
for(A;B;C)
    printf("%d\t",i);
```

Это превращается в

```
A;
while(B)                                    // пока (B)
{
    printf("%d\t",i);
    C;
}
```

Все части есть, но, кажется, цикл с условием продолжения требует больших усилий при вводе. Вы можете сказать, что, если ваши пальцы устали, вы применяете цикл for. Я люблю циклы с условием продолжения, потому что не нужно все запихивать в одну строку. Проще найти отдельные части. Кроме того, есть различные способы управления частью "C" цикла, о которых я расскажу далее.

Другое преимущество цикла с условием продолжения состоит в том, что он выглядит более изящно. Это в особенности справедливо в сравнении с уродливыми конструкциями for(;;). Разговор о них...

Замена неприглядных циклов for(;;) изящными циклами с условием продолжения

Вы помните следующую строку?

```
for(;;)
```

Это цикл for для вечного повторения. Неопределенно долгого. Возможно, он ждет, когда коровы придут домой. Конечно, где-то внутри цикла должен быть оператор завершения break, который быстро его останавливает, когда выполнено некоторое условие. Но проверка выполнения условия для того, чтобы остановить цикл, — разве это не естественная задача для цикла с условием продолжения?

Загрузите программу TYPERS1.C в ваш редактор. Эта программа была представлена в главе 15 "Циклы в языке программирования C". В ней продемонстрировано ключевое слово break. Основная часть программы — следующий цикл for:

```
for(;;)
{
    ch=getchar();
    if(ch=='~')                                // если (ch == '~')
    {
        break;
    }
}
```

Этот пример читается так: “вечно выполняйте следующие инструкции”. Символ `ch` читается с клавиатуры и затем проверяется, была ли введена тильда. Если так (если условие принимает значение `TRUE` (ИСТИНА)), цикл останавливается оператором завершения `break`.

Давайте теперь удалим все эти строки.

Замените их следующими¹:

```
while(ch!='~') // пока (ch!='~')
{
    ch=getchar();
}
```

О, это слишком просто! Инструкции `for` нет, ее заменила инструкция `while`, которая велит повторять следующую инструкцию (или инструкции) до тех пор, пока значение переменной `ch` не станет равным символу тильды. Пока это имеет место, повторяется выполнение данной инструкции. А эта единственная инструкция просто читает символы с клавиатуры и сохраняет их в переменной `ch`.

Сохраните измененный исходный код на диске под именем `TYPER2.C`. Скомпилируйте его и выполните.

После того как вы увидите сообщение `Press ~ then Enter to stop` (Нажмите `<~>`, а затем `<Enter>`, чтобы остановить), напечатайте что-нибудь. Нажмите клавишу тильды, чтобы остановить цикл.

- ✓ Этот пример демонстрирует, как цикл с условием продолжения может изящно заменить уродливый цикл `for`.
- ✓ В этом цикле оператор завершения `break` не нужен, потому что `while` автоматически останавливает цикл, когда `ch` равняется тильде.
- ✓ Условие, проверяемое в данном цикле с условием продолжения, представляет собой отрицание. Это означает, что цикл продолжает выполнение, пока `ch` не равняется символу тильды. Если записать `while(ch=='~')`, цикл повторялся бы только до тех пор, пока пользователь продолжал нажимать тильду на клавиатуре.
- ✓ Операция сравнения *не равняется* обозначается `!=`, т.е. сначала ставится восклицательный знак (в `C` он означает *не*), а за ним — знак `=`.
- ✓ При сравнении символьных констант используются одинарные кавычки.
- ✓ Это не заслуживает вашего внимания, но бесконечный цикл с условием продолжения, эквивалент `for(;;)`, записывается в виде `while(1)`. В этом случае инструкции, принадлежащие циклу, повторяются неопределенно долго или пока оператор завершения `break` не прервет повторение.



Гибкость языка `C`: сокращения в функциональном стиле

Хотя `C` считается строгим языком, он может быть также очень гибким. Вот пример. Предположим, некая функция возвращает значение. Но это совсем не подразумевает, что вы должны сохранить это значение в переменной. Вы можете немедленно использовать значение в другой функции.

В качестве примера рассмотрим функцию `getchar()`, которая возвращает символ, введенный с клавиатуры. Вы можете использовать введенный символ немедленно и не сохранять его в переменной. Это то, что я называю “использованием сокращений в функциональном

¹ Я бы в данном случае такую замену ни за что не посоветовал бы: если переменная `ch` не инициализирована, программа будет неверной и вести себя может непредсказуемо! — Прим. ред.

стиле в языке С². Применение таких сокращений — одно из наиболее впечатляющих свидетельств гибкости языка С.

Код программы TYPER2.C иллюстрирует гибкость языка С. Давайте рассмотрим его фрагмент²:

```
while(ch!='~') // пока (ch!='~')
    ch=getchar();
```

Значение переменной `ch` устанавливается равным результату функции `getchar()`. Иными словами, функция `getchar()` генерирует символ, соответствующий нажатой клавише. Переменная `ch` — только место для хранения, и использование `ch`, чтобы сохранить это значение — просто промежуточный шаг. Чтобы доказать это, я приведу следующий, также вполне законный вариант³:

```
while(getchar()!='~') // пока (getchar()!='~')
    ;
```

Недостаток этого варианта состоит в том, что символ, сгенерированный `getchar()`, нигде не сохраняется. Но это действительно иллюстрирует изящность и языка С, и цикла с условием продолжения.

Заново отредактируйте исходный текст TYPER2.C так, чтобы получилось вот что:

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

```
int main() // главная функция
{
    puts("Start typing."); // Начните печатать
    // Нажмите <~>, а затем <Enter>, чтобы остановиться
    puts("Press ~ then Enter to stop");
    while(getchar() != '~') // пока (getchar() != '~')
        ;
    printf("Thanks!\n"); // Спасибо!
    return(0);
}
```

Теперь не пришлось объявлять переменную `ch`, так что одной строкой стало меньше; не понадобились и фигурные скобки, принадлежащие `while`.

Сохраните измененный файл на диске под именем TYPER3.C. Скомпилируйте и выполните.

Вывод будет тем же самым — ничего удивительного или каких-либо неожиданностей в нем ожидать не приходится. Но код намного более плотный (хотя и напряжение при чтении немного возрастает).



- ✓ Несмотря на эту уловку, часто лучше записать код по шагам, в нескольких строках, а не объединять (не комбинировать) различные операции в одной строке, используя сокращения в функциональном стиле. Разбивая код на несколько шагов, вы облегчаете чтение кода.
- ✓ Сначала расписывайте все выполняемые операции “в полную длину”. Затем, убедившись, что код работает, подумайте о перекодировании с использованием сокращений в функциональном стиле.
- ✓ Если вы действительно используете эту уловку (я подразумеваю сокращения в функциональном стиле), то независимо от длины сокращения, опишите в комментариях, что вы имели в виду. Это поможет позже, при отладке кода.

² Для этого типа цикла крайне важно правильно инициализировать переменную `ch`. Так что это не полноценный эквивалент ранее приведенного цикла `for`. — Прим. ред.

³ Если не считать того, что значение введенного символа теряется, это действительно подлинный эквивалент ранее приведенного цикла `for`. — Прим. ред.

Платить за работу вперед — все равно, что гнать мертвую лошадь...

Вот и пришел черед разобрать вашу первую глупую программистскую шутку на языке C:

```
while(dead_horse)           // пока (за работу платят вперед)
    beat();                 // бить_баклуши();
```

Привлекая ваше обширное знание C, вы можете теперь оценить юмор в клише “бесполезно платить за работу вперед”, оттранслированном на язык программирования C. Вот специальные дополнительные разъяснения:

- ✓ `dead_horse` — название (имя) переменной⁴, значением которой может быть либо TRUE (ИСТИНА), либо FALSE (ЛОЖЬ). Пока `dead_horse` принимает значение TRUE (ИСТИНА), цикл повторяется.
- ✓ Первую строку некоторые программисты расписывают так:

```
while(horse==dead)           // пока (лошадь == мертвая)
    and also                 // а также
    while(!alive)           // пока (!жива)
```

Другими словами, “пока за работу платят вперед...”
- ✓ Функция `beat()` повторяется много раз — все время, пока значение `dead_horse` равно TRUE (ИСТИНА) (или в альтернативной форме — пока лошадь == мертвая (`horse==dead`)).
- ✓ Выражение `beat()` не обязательно помещать в фигурные скобки, потому что это единственная инструкция, принадлежащая циклу.

⁴ Dead horse — это не только мертвая лошадь, но и работа, за которую заплачено вперед. Поэтому клише по use beating a dead horse при тщательном программистском анализе может превратиться в философскую концепцию вроде платить за работу вперед — все равно, что гнать дохлую лошадь. — Прим. ред.

Циклы с условием продолжения. Организация задержки

В этой главе...

- Представление цикла с условием продолжения `do-while`
- Использование цикла задержки
- Вложение циклов
- Создание сетки
- Ключевое слово `continue`

Вы не закончили цикл!

Вы не закончили цикл!

Вы не закончили цикл!

Вы не закончили цикл!

Команды `for` и `while` могут показаться странной парой, и несмотря на то, что они являются официальными ключевыми словами, инициирующими выполнение цикла в языке C, это — не конец истории. Нет, есть несколько способов задания цикла, особенно цикла с условием продолжения. В этой главе рассказывается о причудах циклов и даже демонстрируется несколько ситуаций, в которых необходим мифический “инвертированный” цикл с условием продолжения.

Обычный и инвертированный циклы с условием продолжения

Цикл с условием продолжения может не повториться вообще. Если проверяемое им условие является ложным еще до начала цикла, блок инструкций, предназначенных для повторения, пропускается подобно тому, как вы пропустили бы многие закуски, если бы действительно знали, что находится в них.

Вот холодная, жестокая инструкция `while`:

```
while(v==0)                                // пока (v == 0)
```

Если `v` не равняется 0, инструкции, цепляющиеся за нижнюю сторону цикла с условием продолжения, пропускаются, как если бы вы написали вот что:

```
if(v==0)                                   // если (v == 0)
```

Конечно, есть исключение — своеобразный цикл, который всегда выполняется не менее одного раза. Это инвертированный цикл с условием продолжения, называемый циклом с условием продолжения `do-while`. Этот тип цикла встречается редко, но он имеет очаровательное свойство всегда выполняться хотя бы один раз.

Цикл с условием продолжения `do-while`

Следующая программа содержит простой цикл с условием продолжения, который считает в обратном порядке. Чтобы придать программе некоторый смысл, вы вводите число, с которого начинается отсчет:

```
/* Важная программа для NASA, предназначенная для правильного  
запуска Американского космического корабля. */  
#include <stdio.h>
```

```
int main()                                // главная функция  
{  
    int start;                             // начало  
  
    // Пожалуйста, введите число  
    printf("Please enter the number to start\n");  
    // счет в обратном направлении (1 - 100)  
    printf("the countdown (1 to 100):");  
    scanf("%d",&start);  
    /* Цикл счета в обратном направлении */  
    do  
    {  
        printf("T-minus %d\n",start);  
        start--;  
    }  
    while(start>0);                        // пока (начало > 0)  
  
    printf("Zero!\nBlast off!\n");        // Нуль! Поехали!  
    return(0);  
}
```

Напечатайте этот исходный текст с помощью редактора. Сохраните файл на диске под именем `COUNTDOWN.C`.

Скомпилируйте. Исправьте все ошибки. Обратите внимание, что точка с запятой требуется после конца цикла с условием продолжения `do-while`. Если вы забудете о ней, вы получите сообщение об ошибке.

Выполните программу.

```
Please enter the number to start the countdown (1 to 100):  
Be a traditionalist and type 10. Press Enter:  
T-minus 10  
T-minus 9  
и так далее...  
T-minus 1  
Zero!  
Blast off!
```

(Пожалуйста, введите номер, чтобы начать счет в обратном направлении (1 — 100);
Будьте традиционны и напечатайте 10. Нажмите <Enter>:
T-минус 10
T-минус 9
и так далее...
T-минус 1
Нуль!
Поехали!)



- ✓ Цикл с условием продолжения `do-while` выполняется хотя бы один раз, независимо ни от каких условий.
- ✓ В конце цикла с условием продолжения `do-while` после условия `while` требуется точка с запятой.
- ✓ Перед переменной, используемой в инструкции `scanf`, обязательно нужно поставить амперсанд (&). Без амперсанда программа будет ошибочной.

Цикл `do-while`: подробности с условием продолжения

Цикл с условием продолжения `do-while` имеет только одно преимущество перед традиционным циклом с условием продолжения: он в любом случае хотя бы однажды выполняется.

Ключевое слово `do` принадлежит `while`, как в традиционном цикле с условием продолжения, хотя `while` записывается в самом конце. И единственное отличие состоит в том, что цикл с условием продолжения такого типа всегда выполняется хотя бы однажды, даже когда проверяемое им условие является ложным. Все происходит так, как если бы конструкция `while` даже не обращала внимания на условие до окончания первого выполнения цикла.

Вот стандартный формат цикла этого типа:

```
do                                     // делать
{
    инструкция (и);
}
while(условие);                       // пока (условие)
```

Условие — логическое выражение (обычно неравенство), истинность которого проверяет `while`, — имеет тот же самый вид, что и в `if`. Если условие истинно, повторяются инструкции в цикле. Они продолжают повторяться до тех пор, пока условие не станет ложным, после чего продолжится выполнение остальной части программы. Но независимо от истинности условия инструкции цикла всегда выполняются при первом проходе цикла.

Важно помнить, что после `while` в конце цикла должна стоять точка с запятой. Если забудете, неприятностей не оберетесь.

Цикл с условием продолжения `do-while` имеет несколько, казалось бы, странных особенностей, отличающих его от традиционных циклов `while` и `for`. У этого цикла нет инициализирующей части, `while` вынесено в самый конец, да и инструкции, изменяющие истинность условия, не отмечены особо в его формате. Однако все это легко объяснить. Он не имеет никакого стартового условия, потому что цикл сначала выполняется, и лишь в конце проверяет условие — поэтому-то оно и записано в конце. Несомненно, данный цикл имеет все три части традиционных циклов, фактически, он может выглядеть следующим образом:

```
начало;
do                                     // делать
{
    инструкция (и);
    управление;
}
while(условие);                       // пока (условие);
```

Так что этот вид цикла очень похож на основной цикл с условием продолжения. Это и не удивительно — ведь используется он только тогда, когда что-нибудь должно быть сделано не менее одного раза до проверки условия. Отсюда и его формат, в котором части цикла записаны в другом порядке.

- ✓ Условие, которое проверяет `while`, является логическим выражением, которое принимает значение `TRUE` (ИСТИНА) или `FALSE` (ЛОЖЬ). Вычисление логического выражения выполняется согласно законам С, т.е. точно так же, как и вычисление логического выражения в условном операторе. Вы можете использовать те же символы, которые используются в условии `if`, в том числе и логические операторы (`&&` и `||`), — они подчиняются тем же самым правилам.
- ✓ Этот тип цикла встречается действительно редко (приблизительно 5 процентов всех циклов в С относятся к циклам с условием продолжения типа `do-while`).
- ✓ Чтобы остановить цикл с условием продолжения, можно использовать `break`. Фактически остановить выполнение инструкций внутри цикла может только `break`. Как и в случае циклов `for` и `while`, все инструкции в фигурных скобках повторяются как единый блок, если внутри него нет `break`.

Недостаток программы COUNTDOWN.C

Снова выполните программу `COUNTDOWN`. Когда она попросит ввести число, введите 200.

Вообще-то в этой программе нет ошибки; программа считает в обратном порядке и от 200 до 0. Но, с другой стороны, 200 находится вне диапазона, которому должно принадлежать число, вводимое вами в приглашении.



Используйте полосы прокрутки, чтобы рассмотреть строки, которые пролистала ваша операционная система. Хотелось бы надеяться, что окно с подсказкой DOS (или другое используемое вами окно) имеет полосы прокрутки.

А что, если напечатать 0 или отрицательное число? Пробуйте это теперь; выполните программу и напечатайте -5. Вы увидите вот что:

```
T-minus -5
Zero!
Blast off!
```

```
(Т-минус -5
Нуль!
Поехали!)
```

Цикл с условием продолжения `do-while` выполняется не менее одного раза, поэтому число -5 отображается. Не подумайте, что это один из самых больших ляпов в современной истории программирования. Как раз наоборот, эта программа должна поразить астронавтов, которые лениво ожидают старта, хотя они давно уже должны были взлететь.

Как принять меры против этой бестактности? Нужно записать цикл так, чтобы отсчет запускался только после ввода допустимого значения. Проверку можно выполнить в цикле с условием продолжения. И действительно, с обработкой числа весьма блестяще справляется цикл с условием продолжения `do-while`.

Всегда проверяйте вводимые числа в цикле с условием продолжения do-while



Во всех программах нужно проверять входные данные!

Впрочем, применить этот совет можно лишь в том случае, если вы знаете, какими могут быть допустимые данные. Например, если известен диапазон чисел или символов. В `COUNTDOWN.C` числа должны находиться в диапазоне от 1 до 100. В некоторых программах, например базах дан-

ных, известно, что пользователь может ввести ограниченное количество символов, например 40 или меньше. В общем, должны быть известны какие-либо ограничения вроде указанных выше.

Необходимо гарантировать, что пользователи не смогут заставить программу обрабатывать неправильные, или недопустимые, значения — такие, которые стали бы причиной неправильной работы программы. Вы, например, должны удостовериться, что пользователь напечатал только 40 (или меньше) символов. (Ведь если ввести больше, программа может зависнуть.) Вы должны принять меры против этой ситуации — и вы это можете, если правильно запишете программу.

По традиции, программы этого типа называются *защищенными*, или *пуленепробиваемыми*. Для таких программ защита от данного типа ошибок разрабатывается заранее. Именно поэтому необходимо проверять все типы вводимых пользователем данных, чтобы удостовериться, что они допустимы. В противном случае вы можете попросить повторить ввод снова или напечатать сообщение об ошибках.

Чтобы сделать программу COUNTDOWN.C пуленепробиваемой, необходима программа, которая просит повторить ввод снова всякий раз, когда введенное значение меньше 1 или больше 100. Цикл с условием продолжения `do-while` справляется с этим заданием без труда, и вот почему.

- ✓ Во-первых, цикл с условием продолжения `do-while` выполняет свои операторы хотя бы один раз, независимо от истинности условия. Таким образом, вы можете попросить ввести данные в первый раз, а затем повторить просьбу, если введенное значение находится вне диапазона допустимых значений.
- ✓ Во-вторых, часть цикла `while` выясняет, является ли введенное значение (оно хранится в переменной `start`) меньшим 1 или большим 100.

Для этого в COUNTDOWN.C масштабных изменений не потребуется. Нужно изменить только первую часть программы. Вот так:

```
do
{
    // Пожалуйста, введите число
    printf("Please enter the number to start\n");
    // отсчет в обратном направлении (1 - 100):
    printf("the countdown (1 to 100):");
    scanf("%d",&start);
}
while(start<1 || start>100);           // пока (начало < 1 || начало > 100)
```

Цикл с условием продолжения `do-while` задает тот же самый вопрос, который программа задала ранее. Однако после того как пользователь напечатает значение, часть цикла `while` выясняет, является ли значение переменной `start` меньше 1 ИЛИ больше 100. Если хотя бы одно из этих условий истинно, цикл повторяется, задавая тот же самый вопрос много раз, — пока не будет введено допустимое значение.

Сохраните измененный исходный текст COUNTDOWN.C на диске. Скомпилируйте программу. Выполните ее:

```
Please enter the number to start the countdown (1 to 100):
```

(Пожалуйста, введите число, чтобы запустить отсчет в обратном направлении (1 --- 100):)

Напечатайте 0 и нажмите <Enter>.

Ха! Программа задает вопрос снова. Напечатайте 101 и нажмите <Enter>.

Черт возьми, эта программа действительно умна! Любое значение вне диапазона от 1 до 100 заставляет программу снова и снова просить ввести допустимое число. Это все делает изящный цикл с условием продолжения `do-while`.



- ✓ Проверки диапазонов входных данных, подобные приведенной выше, обязательны для каждой профессиональной программы. Всякий раз, когда пользователь вводит значение в некотором диапазоне или вводит определенное количество символов, цикл проверяет, допустимы ли эти действия пользователя.
- ✓ Большинство проблем в программном обеспечении Microsoft, связанных с критическими или неустранимыми ошибками, вызваны отсутствием этого типа проверки границ.
- ✓ О логическом операторе `| |` (операция OR (ИЛИ)) рассказывается в главе 14 “Логические выражения и ключевое слово `if`”.
- ✓ Вы можете вставить следующий комментарий в исходный текст, чуть выше первого цикла:
/ Этот цикл гарантирует, что пользователь введет надлежащее значение */*

Вложенные циклы и другие глупости

Великолепные циклы внутри циклов (колеса внутри колес), вращающиеся в аттракционе луна-парка, могут усыпить даже контролеров. Но это — другая тема. В языке программирования C вращение двух циклов — обычная и очень полезная на практике вещь, поэтому называются вложенными циклами. Таким образом, когда говорят о вложенных циклах, подразумевают, что один цикл содержится в другом.

Добавление напряженной, драматической задержки к программе COUNTDOWN.C

Что отсутствует в программе COUNTDOWN.C, так это некоторая (неизбежная перед стартом космического корабля) напряженность. Скорость отсчета в обратном направлении в действительности не зависит от введенного значения (от 1 до 100); цифры мелькают на экране. Никакой приостановки!

Чтобы замедлить их отображение, вы можете вставить в программу цикл задержки. Цель цикла задержки состоит в том, чтобы просто занять работой центральный процессор компьютера (чтобы он потреблял временные такты, как двигатель бензин на холостом ходу) — это несколько замедлит программу. Да, вы делаете это нарочно.

Измените второй цикл с условием продолжения `do-while` в программе COUNTDOWN.C так:

```
do
{
    printf("T-minus %d\n",start);
    start--;
    for(delay=0;delay<100000;delay++); /* цикл задержки */
}
while(start>0); // пока (начало > 0);
```

И поскольку программу бы стошнило, если бы переменная `delay` (задержка) не была бы объявлена, добавьте сверху исходного текста следующую строку чуть ниже инструкции `int start`:

```
long delay;
```

Вот полный, модифицированный исходный текст, включая изменения, которые были сделаны в предыдущих разделах:

/ Важная программа для NASA, предназначенная для правильного запуска Американского космического корабля. */*

#include <stdio.h>

```
int main()                                // главная функция
{
    int start;                             // начало
    long delay;                            // задержка

    do
    {
        // Пожалуйста, введите число
        printf("Please enter the number to start\n");
        // счет в обратном направлении (1 - 100)
        printf("the countdown (1 to 100):");
        scanf("%d",&start);
    }
    while(start<1 || start>100);           // пока (начало <1 || начало > 100);

    /* Цикл счета в обратном направлении */

    do
    {
        printf("T-minus %d\n",start);
        start--;
        for(delay=0;delay<100000;delay++); /* цикл задержки */
    }
    while(start>0);                        // пока (начало > 0)

    printf("Zero!\nBlast off!\n");         // Нуль! Поехали!
    return(0);
}
```

Убедитесь, что ваш исходный текст COUNTDOWN.C теперь содержит вложенный цикл для задержки вывода текста на дисплей.

Сохраните исходный текст под именем COUNTDOWN.C. Скомпилируйте и выполните.

Если вывод все равно выполняется слишком быстро, изменяйте значение в цикле **for** от 100 000 до 1 000 000 (записывается так: 1000000). Если даже это не помогает, попробуйте 2 000 000. Если вы хотите указать в качестве значения 4 000 000, то переменную *delay* нужно объявить как **unsigned long** (без знака).

- ✓ Цикл **for** в цикле с условием продолжения служит примером вложенного цикла. Обратите внимание, что оба цикла не обязаны иметь тот же самый тип (например, совсем не обязательно, чтобы оба цикла были циклами **for** или циклами с условием продолжения).
- ✓ Вложенный цикл — это цикл, выполняющийся в другом цикле.
- ✓ Сначала начинает выполняться первый цикл, т.е. внешний цикл. Затем выполняется внутренний цикл, причем внутренний цикл выполняется столько раз, сколько задано. После этого снова продолжается выполнение внешнего цикла, а затем внутренний цикл снова повторяется полностью. Вот так они работают.
- ✓ Каждый цикл имеет свою отдельную переменную, связанную только с одним циклом. Например, следующие два цикла **for** вложены неправильно:

```
for (x=0; x<5; x++)
    for (x=5; x>0; x--);
```





Поскольку *x* используется в обоих циклах, эти вложенные циклы ведут себя не так, как вы ожидаете. Этот цикл фактически бесконечен, потому что оба цикла управляют той же самой переменной, причем пытаются изменять ее в различных направлениях.



- ✓ Подобное бедствие совсем не очевидно в огромных программах, где заготовки двух вложенных циклов могут находиться на расстоянии нескольких миль один от другого. Вы же, не думая об этом, используете вашу любимую переменную *x* (или *i*) в каждом цикле. Подобные ошибки могут испортить вам настроение на весь день.

- ✓ Чтобы избежать порчи вложенных циклов, используйте различные переменные в каждом цикле — например, *a* или *b*, или *i1* и *i2* или даже кое-что осмысленное, типа *start* и *delay*, как в примере вложенных циклов в программе COUNTDOWN.C.

- ✓ Вложенный цикл *for* в COUNTDOWN.C заканчивается точкой с запятой, которая указывает, что он не имеет никаких собственных инструкций, которые должны повторяться. Вот еще один способ форматирования этого цикла:

```
for (delay=0; delay<100000; delay++)  
;
```

Этот пример форматирования показывает, что цикл *for* действительно не имеет никаких инструкций для повторения. Он только занимает время микропроцессора, тратя его впустую (а именно этого вы и хотите).

- ✓ Хотя циклы задержки, такие как в COUNTDOWN.C, являются самыми обычными, существует лучший способ задержки с использованием внутренних часов компьютера для точного указания времени задержки. (Иными словами, этот способ позволяет определить продолжительность задержки.)



- ✓ В связи с программированием циклов задержки я вспоминаю мой первый ПК IBM-PC, купленный приблизительно 20 лет назад. Чтобы добиться от него полусекундной паузы, достаточно было написать цикл задержки, в котором он считал только до 10 000. Современные компьютеры считают намного быстрее — это очевидно!



Время сна!

Язык C действительно имеет встроенную функцию задержки, так что на самом деле программировать цикл задержки не нужно!

Функция *sleep()* приостанавливает программу на заданное количество секунд. Да, именно секунд. Вы указываете число секунд, которые программа должна ждать, внутри круглых скобок функции *sleep()*:

```
sleep(40);
```

Вот так вы можете заставить программу подождать 40 секунд.

Вы можете заменить цикл задержки *for* в COUNTDOWN.C инструкцией

```
sleep(1);
```


Эта строка добавляет драматическую паузу после вывода каждой строки — медленную и часто раздражающую, но инструкция работает.

Обратите внимание, что в некоторых реализациях GCC функция `sleep()`, очевидно, отсчитывает миллисекунды, а не секунды. (Так что параметр этой функции иногда указывает число миллисекунд, а не секунд.). В этом случае, чтобы время задержки было равно, например, одной секунде, в `COUNTDOWN.C` нужно указать следующую команду:

```
sleep(1000);
```

Имейте в виду, что такая реализация функции `sleep()` нестандартна.

Программа GRID.C, полная вложенных циклов

Вложенные циклы встречаются очень часто. Наиболее часто они встречаются при заполнении сеток и массивов. В этом случае вы заполняете ячейки, стоящие на пересечении строк и столбцов. Заполнение вы ведете строку за строкой, или один столбец за другим. Соответствующий пример приведен в программе `GRID.C`, которая отображает сетку из чисел и букв:

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

```
int main()                                // Главная функция
{
    int a;
    char b;

    printf("Here is thy grid...\n");      // Вот ваша сетка...

    for(a=1;a<10;a++)
    {
        for(b='A';b<'K';b++)
        {
            printf("%d-%c ",a,b);
        }
        putchar('\n'); /* конец строки */
    }
    return(0);
}
```

Эта программа создает прямоугольник (хорошо, пусть будет по-вашему — сетку) 10 на 9 из чисел и букв с использованием вложенных циклов.

Введите исходный текст с помощью вашего редактора. Сохраните полученный документ на диске под именем `GRID.C`.

Скомпилируйте программу. Обратите внимание, что компилятор не ругается из-за того, что один цикл `for` вложен в другой (если, конечно, вы не сделали опечатку).

Запустите на выполнение. Вот каким будет вывод:

```
Here is thy grid...
1-A 1-B 1-C 1-D 1-E 1-F 1-G 1-H 1-I 1-J
2-A 2-B 2-C 2-D 2-E 2-F 2-G 2-H 2-I 2-J
3-A 3-B 3-C 3-D 3-E 3-F 3-G 3-H 3-I 3-J
4-A 4-B 4-C 4-D 4-E 4-F 4-G 4-H 4-I 4-J
5-A 5-B 5-C 5-D 5-E 5-F 5-G 5-H 5-I 5-J
6-A 6-B 6-C 6-D 6-E 6-F 6-G 6-H 6-I 6-J
7-A 7-B 7-C 7-D 7-E 7-F 7-G 7-H 7-I 7-J
8-A 8-B 8-C 8-D 8-E 8-F 8-G 8-H 8-I 8-J
9-A 9-B 9-C 9-D 9-E 9-F 9-G 9-H 9-I 9-J
```

(Вот ваша сетка...)

Ничего себе! Такая эффективность должна порадовать любого правительственного бюрократа.

- ✓ Первый, внешний цикл `for` считает от 1 до 10.
- ✓ Внутренний цикл `for` может показаться странным, но это не так. Он только использует в своих интересах двойственную алфавитно-цифровую природу символов в компьютере. Символьная переменная `b` сначала равна символу (букве) `A` и увеличивается на 1 (один символ, точнее, одну букву), пока ее значение меньше символа (буквы) `K`. Так что `b` устанавливается равным значению символа `A` в коде ASCII, которое равно 65. Переменная `b` затем увеличивает свое значение до ASCII-кода символа `K`, которое равно 75. Это немного необычно, но вполне законно.
- ✓ Функция `printf()` отображает числа и символы при выполнении внутреннего цикла. Вы наблюдаете этот процесс на экране: внешний цикл сохраняет значение числа, пока печатаются символы от `A` до `K`. Затем выполняется увеличение во внешнем цикле и печатается следующая строка символов.
- ✓ Обратите внимание, что в функции `printf()` предусмотрен пробел после символа `%c`. Благодаря ему столбцы сетки отделены пробелом друг от друга.
- ✓ Функция `putchar()` отображает отдельный символ на экране. В `GRID.C` она используется для отображения символа перехода на новую строку `\n` в конце каждой строки.

Ключевые слова *break* и *continue*

Для непосредственного управления циклами в программах могут использоваться два ключевых слова языка C. Это ключевые слова `break` и `continue`. Ключевое слово `break` должно быть вам знакомо из главы 15 “Циклы в языке C”; с тех пор оно иногда появлялось в программах. Ключевое слово `continue` — новое, но оно играет столь важную роль, что может понадобиться в дальнейшем.

Ключевое слово `continue` указывает, что нужно немедленно повторить цикл. Оно работает подобно `break`, поскольку остальная часть инструкций в цикле пропускается; но в отличие от `break`, команда `continue` отсылает компьютер назад к началу цикла.

- ✓ И `break`, и `continue` — ключевые слова языка C и инструкции сами по себе. Каждая из них должна заканчиваться точкой с запятой.
- ✓ И `break`, и `continue` можно использовать в любом цикле, лишь бы он был записан на C!
- ✓ Если `break` или `continue` встретится вне цикла, они могут стать причиной весьма неприятных ошибок.

Пожалуйста, продолжайте...

Следующая программа `BORC.C` названа не в честь шведского Повара из Моппет-шоу. Это просто акроним для `break` (прервать), `or` (или) и `continue` (продолжать), потому что именно эти ключевые слова играют важную роль в данной программе:

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

```
int main()
```

```
// главная функция
```

```

{
    int x=0;

    for(;;)
    {
        x++;
        if(x<=5)                                // если (x <=5)
        {
            printf("%d, ",x);
            continue;                            // продолжать
        }
        // (" %d больше 5!\n ", x)
        printf("%d is greater than 5!\n",x);
        break;                                   // прервать
    }
    return(0);
}

```

Напечатайте исходный текст BORC.C с помощью вашего редактора. Сохраните его на диске под именем BORC.C, что означает *break* (прервать) *or* (или) *continue* (продолжать). (Если вы продолжаете думать о поваре из Маппет-шоу, вы сохраните файл на диске под именем BORK.C, а это совсем не то, что нужно.) Скомпилируйте и выполните программу.

Вот распечатка:

1, 2, 3, 4, 5, 6 is greater than 5!

(1, 2, 3, 4, 5, 6 больше, чем 5!)

Программа BORC.C содержит бесконечный цикл *for*. Кроме того, этот цикл сокращен инструкцией *continue*. После того как значение *x* возрастет и станет больше 5, *continue* пропускается и, наконец, оператор завершения *break* останавливает бесконечный цикл *for*.

- ✓ *for(;;)* — бесконечный цикл *for*. Я читаю это как “for ever” (“навсегда”).
- ✓ Обратите внимание на пробел после *%d* и запятую в инструкции *printf()*.
- ✓ Оператор *continue* заставляет пропустить остальную часть цикла — *printf()* и *break*, а затем повторить цикл с начала. Увеличение значения *x* доказывает, что цикл продолжает выполняться.
- ✓ В данной программе *if* не имеет *else*. Дело в том, что *continue* прерывает текущее повторение цикла! Если бы не было *continue*, *break* гарантировало бы, что цикл выполняется только один раз.

Ключевое слово *continue*

Как и ключевое слово *break*, ключевое слово *continue* управляет выполнением (повторением) цикла. Оно заставляет процессор немедленно повторить цикл, перескакивая через оставшиеся инструкции, и начать выполнение цикла с первой строки (той, в которой стоит *for*, *while*, *do* или еще что-нибудь, что может стоять в начале цикла):

```

continue;                                // продолжать

```

Ключевое слово *continue* само по себе представляет инструкцию языка C и потому после него ставится точка с запятой.

Команду *continue* удобно применять тогда, когда в цикле есть инструкции, которые не нужно повторять каждый раз; *continue* позволяет пропустить их и начать цикл сначала.



- ✓ Команду `continue` можно применять в любом цикле, а не только в циклах с условием продолжения.
- ✓ Как и `break`, команда `continue` относится только к циклу, в котором она находится.
- ✓ Ключевое слово `continue` вынуждает компьютер немедленно повторить цикл сначала. Остальная часть инструкций в цикле пропускается, а выполнение инструкций цикла начинается с самого начала.
- ✓ Обратите внимание, что `continue` повторяет инструкции только того цикла, в котором оно находится. Если циклы вложены, то `continue` относится только к одному циклу. Кроме того, `continue` не может повторить цикл с условием продолжения, когда условие цикла ложно.
- ✓ Заметьте, что, хотя `continue` заставляет повторять инструкции цикла, оно не инициализирует цикл повторно. Оно заставляет “выполнять снова”, а не “начать заново”.
- ✓ Относительно команды `continue` вы должны иметь в виду только два серьезных предупреждения: не используйте ее вне цикла и не надейтесь, что она воздействует на циклы, внешние по отношению к тому, в который она вложена; внимательно следите за тем, в какое место цикла с условием продолжения вы помещаете ее, чтобы вы не перескочили через счетчик цикла и не создали случайно бесконечный цикл, так как в этом случае программа просто заикнется, и ошибку будет найти весьма непросто.
- ✓ В качестве заключительного утешения отмечу, что эта команда используется редко. На самом деле многие программисты, пишущие на языке C, далеко не всегда знают точно, как применять эту команду.

Разбор случаев в языке C: переключатель switch-case

В этой главе...

- Решение проблемы бесконечного else-if
- Использование переключателя switch-case для разбора случаев
- Создание конструкции выбора: переключатель switch-case

Честно говоря, я не считаю, что переключатель switch-case на самом деле является разновидностью цикла. Но слово *цикл* подходит для него гораздо лучше, чем альтернативный термин, *конструкция выбора*. Дело в том, что инструкции, содержащиеся в конструкции выбора (в переключателе) в действительности не повторяются, но все же они часто используются внутри цикла. И поэтому они, так или иначе, связаны с циклами.

В этой главе рассказывается о последней конструкции, связанной с циклами в языке C, называемой конструкцией выбора, или переключателем switch-case. Это совсем не цикл, но данная замечательная конструкция позволяет решить проблему многочисленных условных операторов. Как и большинство других конструкций в языках программирования, ее лучше просто сначала показать на примере, чем пробовать объяснить с помощью заумных терминов. Именно этому и посвящена данная глава.

*Эти хитрые циклы — переключатели
switch-case*

*Let's all go to the lobby,
Let's all go to the lobby,
Let's all go to the lobby,
And get ourselves a treat!*

Неизвестный автор¹

И когда вы добираетесь до буфета, вы заказываете себе всевозможные вкусняшки из меню. Фактически, менеджеры в вашем местном театре только что изобрели интересную компьютерную программу, которая помогает сократить этих противных ужасных служащих с почасовой заработной платой. Вот изобретенная ими программа:

```
/* Программа для закуской в фойе театра */  
#include <stdio.h>
```

¹ Давайте же войдем в фойе, Давайте же войдем в фойе, Давайте же войдем в фойе, Давайте же войдем в фойе, И угостимся там на славу! (в фойе, как вы понимаете, имеется буфет). — Прим. ред.

```

int main()                                // главная функция
{
    char c;

    // Пожалуйста, сделайте ваш выбор
    printf("Please make your treat selection:\n");
    printf("1 - Beverage.\n");           // 1 - Напиток
    printf("2 - Candy.\n");              // 2 - Леденец
    printf("3 - Hot dog.\n");            // 3 - Хот-дог
    printf("4 - Popcorn.\n");            // 4 - Жареная кукуруза (попкорн)
    printf("Your choice:");              // Ваш выбор:

    /* Выяснить, что они напечатали. */

    c=getchar();
    if(c=='1')
    // 1 - Напиток. Это будет $8.00
        printf("Beverage\nThat will be $8.00\n");
    else if(c=='2')
    // 2 - Леденец. Это будет $5.50
        printf("Candy\nThat will be $5.50\n");
    // 3 - Хот-дог. Это будет $10.00
    else if(c=='3') printf("Hot dog\nThat will be $10.00\n");
    // 4 - Жареная кукуруза (попкорн). Это будет $7.50
    else if(c=='4') printf("Popcorn\nThat will be $7.50\n");
    else
    {
    // неправильный выбор
        printf("That is not a proper selection.\n");
    // я предполагаю, что вы совсем не голодны
        printf("I'll assume you're just not hungry.\n");
    // я могу помочь следующему в очереди?
        printf("Can I help whoever's next?\n");
    }
    return(0);
}

```

Напечатайте этот исходный текст с помощью вашего редактора. Сохраните его на диске под именем LOBBY1.C. Что касается 1 в конце имени файла, то благодаря ей вы знаете, что задача угощения в театральном буфете достаточно серьезна, и ей будут посвящены еще несколько программ LOBBY...

Скомпилируйте. Исправьте все ошибки. Конечно, несколько ошибок могут вкратиться, потому что это такая длинная программа. Соблюдайте правила правописания и помните о точках с запятой. Перетранслируйте после исправления всех ошибок.

Запустите на выполнение:

```

Please make your treat selection:
1 - Beverage.
2 - Candy.
3 - Hot dog.
4 - Popcorn.
Your choice:

```

(Пожалуйста, сделайте ваш выбор:

```

1 — напиток.
2 — леденец.
3 — хот-дог.
4 — жареная кукуруза
Ваш выбор:)

```

Нажмите 2, выбирая Candy (леденец). Любите горячее тамале? Вы увидите:

Candy
That will be \$5.50

(Леденец
Это будет 5,50 \$)

Ну и цены! За это китайское горячее тамале? В следующий раз я украду еду...

Выполните программу снова и попробуйте еще несколько вариантов. Затем попробуйте выбрать опцию, отсутствующую в списке. Напечатайте M (пусть это значит Маргарита):

That is not a proper selection.
I'll assume you're just not hungry.
Can I help whoever's next?

(Это — не надлежащий выбор.
Предполагаю, что вы совсем не голодны.
Я могу помочь следующему в очереди?)
Ладно.

Переключатель switch-case в программе LOBBY

Не кажется ли вам, что смотрятся все эти else-if в программе LOBBY1.C немного забавно? Разве вся эта конструкция не кажется неуклюжей? Возможно, нет. Но все-таки она довольно неуклюжа. Дело в том, что в C есть лучший способ выбрать один из нескольких вариантов. Здесь поможет специальный цикл — переключатель switch-case.

Я должен вам сразу же сказать, что переключатель switch-case — на самом деле совсем не цикл. Да, не цикл, а *инструкция выбора*, которая официально как раз и обозначает то, что делает условный оператор. Переключатель switch-case позволяет выбирать один из нескольких элементов подобно тому, как это делает длинная, сложная последовательность условных операторов того вида, который мы встретили в программе LOBBY1.C.

Далее приведен исходный текст LOBBY2.C, внутреннее (существенное с точки зрения программиста, но незаметное для пользователя) усовершенствование программы LOBBY1.C. Внутреннее, потому что здесь изменился только внутренний вид программы, она стала изящнее. Внешне же программа работает так же, как и раньше:

```
/* Программа для закускойной в фойе театра */
#include <stdio.h>

int main()                                // главная функция
{
    char c;

    // Пожалуйста, сделайте ваш выбор
    printf("Please make your treat selection:\n");
    printf("1 - Beverage.\n");           // 1 - Напиток
    printf("2 - Candy.\n");               // 2 - Леденец
    printf("3 - Hot dog.\n");             // 3 - Хот-дог
    printf("4 - Popcorn.\n");             // 4 - Жареная кукуруза (попкорн)
    printf("Your choice:");               // Ваш выбор:

    /* Выяснить, что они напечатали. */
```

```

c=getchar();
switch(c)                                     // переключатель
{
    case '1':                                // случай '1':
// 1 - Напиток. Это будет $8.00
    printf("Beverage\nThat will be $8.00\n");
    break;                                    // прервать
    case '2':                                // случай '2':
// 2 - Леденец. Это будет $5.50
    printf("Candy\nThat will be $5.50\n");
    break;                                    // прервать
    case '3':                                // случай '3':
// 3 - Хот-дог. Это будет $10.00
    printf("Hot dog\nThat will be $10.00\n");
    break;                                    // прервать
    case '4':                                // случай '4':
// 4 - Жареная кукуруза (попкорн). Это будет $7.50
    printf("Popcorn\nThat will be $7.50\n");
    break;                                    // прервать
    default:                                 // по умолчанию:
// неправильный выбор
    printf("That is not a proper selection.\n");
// Предполагаю, что вы совсем не голодны
    printf("I'll assume you're just not hungry.\n");
// я могу помочь следующему в очереди?
    printf("Can I help whoever's next?\n");
}
return(0);
}

```

Сохраните программу LOBBY1.C в вашем редакторе. Используйте ее исходный текст для LOBBY2.C как основу для редактирования.

Вы заменили все условные операторы переключателем switch-case. Будьте внимательны при вводе. Когда закончите, перепроверьте строки исходного текста после ключевого слова switch, чтобы удостовериться, что вы ввели программу правильно. (Несколько первых строк программы не изменяются.) Скомпилируйте. Исправьте все ошибки и опечатки. Обратите внимание на двоеточия — это не точки с запятой! — в строках с ключевым словом case. Символьные константы заключены в одинарные кавычки. После слова default (по умолчанию) также стоит двоеточие.

Выполните. Вы не увидите никакого отличия в выводе. Внутренне, однако, вы преобразовали уродливую последовательность инструкций else-if в изящную конструкцию принятия решений — в цикл switch-case (переключатель, или конструкцию выбора).

- ✓ Подробно о том, как работает цикл switch-case, рассказывается в следующем разделе.
- ✓ Команда switch сравнивает один символ, введенный с клавиатуры (в предшествующей ей строке), и ищет совпадение с символами в различных операторах выбора в ее фигурных скобках.
- ✓ Каждый из операторов выбора (строки с ключевым словом case) сравнивает свою символьную константу со значением переменной c. Если есть совпадение, выполняются инструкции, принадлежащие данному case (случаю).

- ✓ Инструкция `break` в каждом операторе выбора (`break` после ключевого слова `case`) говорит компилятору, что данный переключатель `switch-case` выполнен, и остальную часть инструкций нужно пропустить.
- ✓ Если `break` отсутствует, выполняется следующая группа операторов выбора, которые следуют после следующего `case`. В этом случае они выполняются независимо ни от чего, поэтому не пропускайте `break`! Отсутствие `break` — орава для циклов типа `switch-case` (для переключателей).
- ✓ Конечный элемент в переключателе `switch-case` — `default` (по умолчанию). Эта опция выполняется тогда, когда совпадение не будет найдено.

Старый трюк с переключателем `switch-case`

Переключатель `switch-case` удобно применять во многих случаях. Но, несмотря на важность и удобство применения переключателя `switch-case`, в программе он выглядит настолько изящно, что многие новички просто в упор не замечают его. Мой совет следующий. Тщательно разберите все программы из этой главы, чтобы почувствовать это изящество и затем внимательно прочитайте в этом разделе абзацы, помеченные галочками, чтобы еще раз повторить ключевые пункты или выяснить то, на что вы не обратили внимания.

Ключевое слово `switch` позволяет в программе самым простым способом выбрать одну из альтернатив. Им нужно заменить любую длинную последовательность повторяющихся инструкций `if-else`, используемых для выполнения операций сравнения.

Ключевое слово `switch` позволяет создать сложную конструкцию, которая включает ключевые слова `case`, `break` и `default`. Вот как она может выглядеть:

```
switch(выбор)
{
    case item1:                // случай item1:
        инструкция (и);
        break;                // прервать
    case item2:                // случай item2:
    case item3:                // случай item3:
        инструкция (и);
        break;                // прервать
    default:                 // по умолчанию:
        инструкция (и);
}
```

Здесь *выбор* обозначает переменную. Она может обозначать клавишу, нажатую на клавиатуре, значение, возвращенное мышью или джойстиком, или некоторое число или символ, который программа должна сравнить с заданными значениями.

После ключевого слова `case` следуют различные *элементы*: `item1`, `item2`, `item3` и так далее — это различные элементы, с которыми сравнивается переменная *выбор*. Каждый из них — это константа, символ или значение; *они не могут быть переменными*. Кончается строка с ключевым словом `case` двоеточием, а не точкой с запятой.

Каждому элементу ключевого слова `case` принадлежит одна или несколько *инструкций*. Программа выполняет *эти инструкции*, когда элемент совпадает со значением переменной *выбор*, — в этом и состоит принятие решения с помощью переключателя `switch`; конечно, это подобно тому, что делает условный оператор `if`, если его условие удовлетворяется. *Инструкции не* заключаются в фигурные скобки. Более того, *инструкций* может не быть вообще. Последней инструкцией в группе операторов выбора `case` обычно бывает команда `break`. Если `break` отсутствует, программа продолжает инструкции в следующем операторе выбора `case`.

Последний элемент в конструкции переключателя `switch` — `default`. Он содержит инструкции, которые будут выполнены, когда совпадение не будет найдено — в этом он подобен заключительному `else` в конструкции `if-else`. Заданные по умолчанию инструкции выполняются независимо ни от чего — если, конечно, вы не покинете конструкцию с помощью `break`.

Чрезвычайно важно помнить о переключателе `switch-case` вот что: программа всегда полностью выполняет все инструкции до `break`, поэтому обязательно вставляйте `break` там, где нужно остановить выполнение переключателя `switch-case`. Рассмотрим, например, следующий фрагмент программы:

```
switch(key)
{
    case 'A':                // случай 'A':
        printf("The A key.\n"); // клавиша A
        break;              // прервать
    case 'B':                // случай 'B':
        printf("The B key.\n"); // клавиша B
        break;              // прервать
    case 'C':                // случай 'C':
    case 'D':                // случай 'D':
        printf("The C or D keys.\n"); // C или D
        break;              // прервать
    default:                 // по умолчанию:
        // я не знаю эту клавишу
        printf("I don't know that key.\n");
}
```

Предположим, что `key` (клавиша) — односимвольная переменная, содержащая символ, который был только что напечатан на клавиатуре. Вот три примера того, как работал бы приведенный выше переключатель:

Пример 1. Предположим, что переменная `key` содержит символ `A`. Программа работает так:

Выбираем значение переменной `key`! Так, `key` равняется прописной (большой) букве `A`. Идем вниз по списку `case`:

```
case 'A':                // случай 'A':
```

Да, мы нашли совпадение. Значение переменной `key` равняется константе, прописной (большой) букве `A`. Выполняем инструкции:

```
printf("The A key.\n"); // клавиша A
```

Печатаем сообщение. Затем:

```
break;                  // прервать
```

Завершаем выполнение переключателя `switch-case`. Все сделано.

Если бы вы не поставили `break` в этом месте, остальные инструкции переключателя (конструкции выбора `switch-case`) были бы выполнены *независимо ни от чего*.

Пример 2. Предположим, что пользователь нажал клавишу `<C>`. Вот как работает программа в этом случае:

```
switch(key)
```

Теперь нужно проверять операторы выбора `case` на предмет поиска совпадения:

```
case 'A': // случай 'A':
```

Нет! Переходим к следующему `case`:

```
case 'B': // случай 'B':
```

Нет! Перескакиваем к следующему `case`:

```
case 'C': // случай 'C':
```

Да! Совпадение есть! Значение переменной `key` равняется символьной константе `'C'`. Что идет дальше?

```
case 'D': // случай 'D':
```

Компьютер только радуется. Нет никакого совпадения, но он все равно ждет команд, потому что он нашел совпадение для `case 'C'`. Поскольку первыми инструкциями, которые он находит, оказываются инструкции после `case 'D'`, он выполняет их:

```
printf("The C or D keys.\n"); // C или D
```

Печатаем сообщение. Затем:

```
break; // прервать
```

Остальная часть конструкции выбора (переключателя `switch-case`) пропускается.

Пример 3. На сей раз с клавиатуры вводится символ `X`. Вот как переключатель `switch-case` работает в этом случае:

```
switch(key)
```

Значение переменной `key` равно `X`. Компьютер ищет соответствующий оператор выбора `case`:

```
case 'A': // случай 'A':
```

Нет! Перескакивает к следующему оператору `case`:

```
case 'B': // случай 'B':
```

Нет! Перескакивает к следующему оператору `case`:

```
case 'C': // случай 'C':
```

Нет! Перескакивает к следующему оператору `case`:

```
case 'D': // случай 'D':
```

Нет! Все случаи просмотрены. Остался только элемент `default`, который может обрабатывать все остальное — включая и `X`:

```
default:
```

и единственная инструкция:

```
// я не знаю эту клавишу
printf("I don't know that key.\n");
```

Выполнение конструкции выбора (переключателя `switch-case`) закончено.

- ✓ Значение переменной *выбор* в круглых скобках переключателя `switch` должно быть символом или целым числом. Большинство программистов указывает там символьную или целочисленную переменную. Но в круглых скобках можно поместить

также инструкцию языка C или функцию, лишь бы ее значение после вычисления было символом или целым числом.

- ✓ Строка, в которой находится `case`, заканчивается двоеточием, а не точкой с запятой. Инструкции, принадлежащие блоку `case`, не заключаются в фигурные скобки.
- ✓ Последней инструкцией, принадлежащей группе операторов выбора `case`, обычно является `break`. Эта инструкция заставляет компьютер перескочить через остальную часть переключателя (конструкцию `switch`) и продолжить выполнение программы.
- ✓ Если забыть `break`, остальная часть переключателя (конструкции `switch`) продолжает выполняться. Это может быть не совсем то, что нужно.
- ✓ Компьютер сравнивает каждый элемент в операторе выбора `case` со значением переменной *выбор* — это и есть задача переключателя `switch`. Если совпадение найдено, выполняются инструкции, принадлежащие найденному блоку `case`, в противном случае они пропускаются.
- ✓ Иногда в `case` нет никаких инструкций. Тогда в случае совпадения программа просто “проваливается” к следующему оператору выбора `case`.
- ✓ После ключевого слова `case` должно следовать постоянное значение — число или символ. Например:

```
case 56: /* выбран элемент 56 */           // случай 56:
```

или

```
case 'L': /* нажата клавиша L */           // случай 'L':
```

Однако переменную после ключевого слова `case` указать нельзя. Этот фокус просто не пройдет. Даже если очень хочется. Вы можете даже послать мне письмо по электронной почте с вопросом, можете ли вы это сделать, но это все равно не поможет. Даже не мечтайте об этом.

- ✓ В большинстве руководств по C вся команда `switch` называется переключателем, а `case` рассматривается только как еще одно ключевое слово. Я рассматриваю конструкцию `switch-case` как модуль, потому что это помогает мне помнить, что второе слово — `case`, а не что-нибудь другое.
- ✓ В конце конструкции `switch-case` ключевое слово `default` не обязательно. Если его нет и не будет найдено ни одного совпадения с каким-либо из элементов в `case`, никакие инструкции не выполняются.

Переключатель switch-case в циклах с условием продолжения while

Ядро большинства программ составляет цикл с условием продолжения `while`. Внутри этого цикла с условием продолжения обычно находится хорошая, большая конструкция выбора — переключатель `switch-case`. Дело в том, что вы можете много раз выбирать варианты, пока не выберете опцию, которая закрывает программу. Много раз — это цикл `while`, а выбор делается с помощью конструкции выбора — переключателя `switch-case`. Чтобы разъяснить все это подробнее, я представляю заключительную реализацию программы LOBBY.C:

```

/* Программа для закуской в фойе театра */
#include <stdio.h>

int main()                                // главная функция
{
    char c;
    int done;
    float total=0;

// Пожалуйста, выберите себе продукты
printf("Please make your treat selections:\n");
printf("1 - Beverage.\n");           // 1 - Напиток
printf("2 - Candy.\n");               // 2 - Леденец
printf("3 - Hot dog.\n");             // 3 - Хот-дог
printf("4 - Popcorn.\n");            // 4 - Жареная кукуруза (попкорн)
printf("= - Done.\n");               // Сделано
printf("Your choices:");              // Вы выбрали:

/* Выяснить, что они напечатали. */

done=0;
while(!done)
{
    c=getchar();

    switch(c)
    {
        case '1':                    // случай '1':
// 1 - Напиток.
printf("Beverage\t\t$8.00\n");
total+=8;                            // общее количество += 8
break;                               // прервать
        case '2':                    // случай '2':
// 2 - Леденец.
printf("Candy\t\t\t$5.50\n");
total+=5.5;                          // общее количество += 5.5
break;                               // прервать
        case '3':                    // случай '3':
// 3 - Хот-дог.
printf("Hot dog\t\t\t$10.00\n");
total+=10;                           // общее количество += 10
break;                               // прервать
        case '4':                    // случай '4':
// 4 - Жареная кукуруза (попкорн).
printf("Popcorn\t\t\t$7.50\n");
total+=7.5;                          // общее количество += 7.5
break;                               // прервать
        case '=':                    // случай '=':
// (" = Общее количество $ %.2f\n ", общее количество)
printf("= Total of $%.2f\n",total);
// Пожалуйста, заплатите кассиру
printf("Please pay the cashier.\n");
done=1;
break;                               // прервать
        default:                     // по умолчанию
// Неподходящий выбор
printf("Improper selection.\n");

```

```

    } /* конец переключателя switch */
} /* конец цикла с условием продолжения while (пока) */
return(0);
)

```

Напечатайте исходный текст LOBBY3.C с помощью вашего редактора. Вы можете попробовать отредактировать программу LOBBY2.C, но я сделал много тонких изменений в программе и не хотел бы, чтобы вы пропустили хоть одно из них. Начните с чистого листа.

Сохраните файл на диске под именем LOBBY3.C.

Скомпилируйте. Исправьте все ошибки, которые, возможно, проскользнули в код.

Запустите на выполнение:

Please make your treat selection:

```

1 - Beverage.
2 - Candy.
3 - Hot dog.
4 - Popcorn.
= - Done.

```

Your choice:

(Пожалуйста, выберите себе продукты:

```

1 — Напиток.
2 — Леденец.
3 — Хот-дог.
4 — Жареная кукуруза (попкорн).
= — Сделано.

```

Вы выбрали:)

Чтобы должным образом выполнить программу, сразу напечатайте все выбранные вами продукты, после чего нажмите клавишу со знаком "=" и затем <Enter>. Эта стратегия позволяет избежать проблемы буферизации в функции `getchar()`, о которой рассказывалось ранее, но которая находится вне контекста данной главы.

Например, если вы хотите напиток и хот-дог, напечатайте **13=** и затем нажмите <Enter>.

```

Beverage $8.00
Hot dog $10.00
Total of $18.00
Please pay the cashier.

```

(Напиток 8,00 \$

Хот-дог 10,00 \$

Общее количество 18,00 \$

Пожалуйста, заплатите кассиру.)

Выполните программу снова, введите **123412341234xx92431=**, а затем нажмите <Enter>:

```

Beverage $8.00
Candy $5.50
Hot dog $10.00
Popcorn $7.50
Beverage $8.00
Candy $5.50
Hot dog $10.00
Popcorn $7.50
Beverage $8.00
Candy $5.50
Hot dog $10.00
Popcorn $7.50
Improper selection.

```

Improper selection.
Improper selection.
Candy \$5.50
Popcorn \$7.50
Hot dog \$10.00
Beverage \$8.00
Total of \$124.00
Please pay the cashier.

(Напиток 8,00 \$
Леденец 5,50 \$
Хот-дог 10,00 \$
Жареная кукуруза (попкорн) 7,50 \$
Напиток 8,00 \$
Леденец 5,50 \$
Хот-дог 10,00 \$
Жареная кукуруза (попкорн) 7,50 \$
Напиток 8,00 \$
Леденец 5,50 \$
Хот-дог 10,00 \$
Жареная кукуруза (попкорн) 7,50 \$
Неподходящий выбор.
Неподходящий выбор.
Неподходящий выбор.
Леденец 5,50 \$
Жареная кукуруза (попкорн) 7,50 \$
Хот-дог 10,00 \$
Напиток 8,00 \$
Общее количество 124,00 \$
Пожалуйста, заплатите кассиру.)

Последний раз я угощаю вас в театральном буфете!

- ✓ В большинстве программ используется именно этот тип цикла. Цикл с условием продолжения `while(!done)` выполняется до тех пор, пока конструкция `switch-case` не обработает все вводимые в программу символы.
- ✓ Один из элементов конструкции `switch-case` обрабатывает условие, при котором цикл должен остановиться. В `LOBBY3.C` это нажатие клавиши со знаком “=”. Это событие устанавливает значение переменной `done` равным 1. В результате прекращается повторение цикла с условием продолжения.
- ✓ В `C` значение 0 рассматривается как `FALSE` (ЛОЖЬ). Устанавливая переменную `done` равной 0, с помощью знака ! (не) получаем отрицание ее значения. Используя полученное значение в качестве условия, выполняем повторение цикла с условием продолжения. Причина всего этого состоит в том, что цикл `while(!done)` можно прочесть как “пока не сделано” (“while not done” по-английски).
- ✓ Различные конструкции выбора (точнее, ключевые слова `case`) затем проверяют, какая клавиша была нажата. Для каждого совпадения в случаях 1–4 выполняются три инструкции: заказанный элемент (продукт) отображается на экране; общее количество (денег) увеличивается на стоимость заказанного элемента (выполняется инструкция вроде `total+=3`; и наконец, выполняется оператор завершения `break`

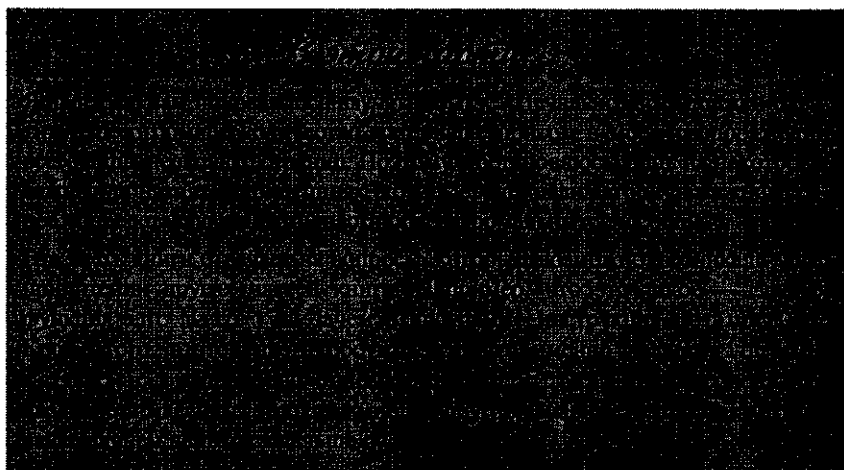
конструкции `switch-case`. Затем цикл с условием продолжения повторяется далее, чтобы дать возможность покупателю выбрать дополнительные продукты.

- ✓ Об операторе `+=` рассказывалось в главе 16 “Знакомство с циклами и применением инкремента (оператора `++`) в циклах”. Например, `total += value` (общее количество `+=` значение) — сокращение для `total = total + value` (общее количество = общее количество + значение).

Часть IV

Язык С: следующий уровень





Создание первой функции

В этой главе...

- Понятие функции
- Создание функции `jerk()`
- Прототипы функций
- Как обойтись без прототипа (располагать функции снизу вверх)
- Формат функций и их именование

Ф

ункции — это способ определить ваш собственный язык с помощью языка C. Они — изящные небольшие процедуры, или ряд команд, которые велят компьютеру делать то, что вы хотите. Все эти команды как бы связаны в один пакет, который удобно использовать в программе, притом, возможно, неоднократно. В некотором смысле, создание функций походит на добавление ваших собственных команд к языку C.

Если вы знакомы с языками программирования, вы должны знать, что функции подобны подпрограммам или процедурам. Если вы не знакомы с программированием, представляйте функции в виде сокращений. Функция — “черный ящик”, который делает нечто замечательное или таинственное. Созданную вами функцию можно использовать в остальной части вашей программы точно так же, как любую другую функцию языка C или ключевое слово.

(Заметьте, что в заголовке этой главы отсутствует упоминание языка C. Да, мне тоже немало надоело C, пора уже научиться определять собственный язык.)

Знакомство с функциями

Действительно ли функции необходимы? Абсолютно! Каждая программа должна иметь не менее одной функции — а именно главную функцию `main()`. Это неизбежно. Кроме нее, вы не обязаны создавать ваши собственные функции. Но без них код, как правило, содержит много повторяющихся команд.

Глупый пример программы, которую вам не придется вводить

Предположим, что вы пишете программу, которая запускает проигрывание песни “Alexander’s Rag Time Band” каждый раз, когда пользователь делает что-то приятное. Поскольку программа, требуемая для запуска исполнения песни, немного необычна и уникальна, вы решаете написать функцию, которая превращает ПК в оркестр. Шутки ради предположим, что такая функция напоминает следующий код:

```
playAlex()  
{  
    play(466,125);           // играть (466,125);  
    play(494,375);           // играть (494,375);  
    play(466,125);           // играть (466,125);  
}
```

```

    play(494,1000); // играть (494,1000);
    /* и еще много других подобных строк ... */
}

```

Предположим, что функция `play()` (играть()) заставляет динамик компьютера издавать определенный тон в течение некоторого времени. Тогда множество функций `play()` сможет заставить компьютер насвистывать мелодию.

Если все эти инструкции аккуратно упаковать в функцию, программа сможет “вызывать” эту функцию каждый раз, когда нужно будет начать исполнение песни. Вместо того чтобы повторно записывать огромное количество строк-команд, вы просто пишете следующую строку в вашем коде:

```
playAlex();
```

Это — “команда” языка C. Она гласит: “иди туда, где находится функция `playAlex()`, и делай то, что там написано, а затем возвращайся сюда, чтобы продолжить выполнение программы”. Оказывается, функции могут упростить запись программы.

- ✓ Чтобы создать функцию, ее нужно записать в исходном тексте. Функция имеет название (имя), всякие дополнительные безделушки, а также свой собственный код на языке C, который выполняет задачу функции. Подробнее обо всем этом я расскажу позже в этой главе.
- ✓ Чтобы использовать функцию, ее нужно вызвать. Йо-хо-хо! Чтобы вызвать функцию, нужно в программе напечатать название (имя) функции, а за ним — круглые скобки (они могут быть пустыми):

```
playAlex();
```

 Эта команда вызывает функцию `playAlex()`, и компьютер уходит и делает то, что написано в этой функции, т.е. выполняет инструкции, записанные в коде этой функции.
- ✓ Да, это правильно. Вызвать (или использовать) функцию столь же просто, как вставить ее название (имя) в исходный текст. Иными словами, вызов функции делается точно так же, как и любой другой инструкции языка C.
- ✓ Некоторым функциям требуется передать информацию в их круглых скобках. Например, такой функцией является `puts`:

```
// О, сколько величественных способов начать день!
puts("Oh, what a great way to start the day.");
```
- ✓ Некоторые функции возвращают значение, т.е. они возвращают нечто такое, что ваша программа может использовать, исследовать, сравнивать или выполнять другие операции над ним. Функция `getchar()` возвращает символ, напечатанный на клавиатуре. Этот символ обычно сохраняется в символьной переменной; вот пример:

```
thus=getchar();
```
- ✓ Некоторые функции требуют, чтобы в круглых скобках передавались параметры, и возвращают значение.
- ✓ Другие функции (например, `playAlex()`) не обязаны ни получать параметры в круглых скобках, ни возвращать значение.
- ✓ С функциями вы на самом деле хорошо знакомы. В большинстве программ на C функции используются вовсю, вспомните хотя бы о функциях `printf()`, `getchar()`, `atoi()` и других. В отличие от функций, написанных прикладным программистом, эти функции входят в библиотеку функций, поставляемой вместе с компилятором. Но даже такие функции работают по тем же правилам, что и написанные прикладным программистом.



- ✓ Создавая функцию, программист на самом деле не добавляет новое слово к языку C. Однако функцию, написанную вами, вы можете использовать в ваших программах точно так же, как и любую другую функцию, например `printf()` и `scanf()`.

Как избежать излишних повторений с помощью функций

Я люблю с помощью функций избавляться от излишних повторений инструкций в программах. Если что-нибудь программа делает более одного раза, перенесите выполняемые действия в функцию. Это значительно упрощает запись программы и помогает разбить главную функцию (`main()`) на вызовы функций, что может сократить исходный текст (который может стать утомительно длинным).

Следующий пример — программа `BIGJERK1.C`, унылый перечень проклятий безобразнику Биллу, который звонит трижды в день:

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

```
int main() // главная функция
{
    // Он звонит мне по телефону и ничего не говорит
    printf("He calls me on the phone with nothing say\n");
    // Не раз, не два, а три раза в день!
    printf("Not once, or twice, but three times a day!\n");
    printf("Bill is a jerk!\n"); // Билл - безобразник!
    // Он оскорбляет мою жену, моего кота, мою мать
    printf("He insulted my wife, my cat, my mother\n");
    // Он раздражает и морочит голову, как никто другой!
    printf("He irritates and grates, like no other!\n");
    printf("Bill is a jerk!\n"); // Билл - безобразник!
    // Он хихикает, этот толстяк
    printf("He chuckles it off, his big belly a-heavin'\n");
    // Но он не будет смеяться, когда я доберусь до него!
    printf("But he won't be laughing when I get even!\n");
    printf("Bill is a jerk!\n"); // Билл - безобразник!
    return(0);
}
```

Введите эту программу с помощью вашего редактора. Перепроверьте все круглые скобки и двойные кавычки. Они раздражают. Раздражают, но они нужны!

Скомпилируйте и выполните `BIGJERK1.C`. Эта программа отображает на экране унылый перечень проклятий. Унылое ворчание. Ничего больше. Обратите внимание, что один фрагмент программы повторяется три раза. Кажется, можно использовать функции.

- ✓ Большинство повторений, которых позволяет избежать функция, намного более сложны, чем простая инструкция `printf()`. Ах, но это ведь только демонстрационная версия.
- ✓ Ни один из тех Биллов, с которыми я знаком лично, не является безобразником. Можете заменить имя `Bill` другим именем, если вы чувствуете (убеждены), что оно больше подходит для этой песенки.
- ✓ В былые дни (когда я был совсем молод) каждый байт в программе был на счету. Сообщения вроде `Bill is a jerk!` (Билл — безобразник!) повторяются много раз. А это значит, что драгоценные байты данных были потрачены впустую на глупую строку текста. Древние программисты, вроде меня, оттачивали свои навыки, освобождая дополнительные байты в программах для более существенной информации, — они экономили на сообщениях, по-



добных приведенному выше. Сокращение размера программы с 4 096 байтов до 3 788 байтов считали выдающимся достижением. Конечно, на сегодняшних суперкомпьютерах такая экономия памяти считается несущественной.

Благородная функция `jerk()`, позволяющая устранить повторение безобразий в исходном коде

Пришло время добавить ваше первое новое слово к языку C — функцию `jerk()`. Хорошо, `jerk` — не слово языка C. Это — функция. Но вы используете в программе ее имя так же, как вы использовали бы любое другое слово языка C или функцию. Компилятор не различает их, если вы делаете все должным образом.

Теперь мы можем написать новую, улучшенную программу, поющую “Bill is a jerk” (“Билл — безобразник”). Эта программа содержит благородную функцию `jerk()`, прямо рядом с основной (главной) функцией `main()`. Эта программа — главный шаг в совершенствовании ваших навыков программирования. Итак, можете считать, что наступил исторический момент. Сделайте паузу, чтобы насладиться результатом после ввода программы:

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

```
int main()                                // главная функция
{
    // Он звонит мне по телефону и ничего не говорит
    printf("He calls me on the phone with nothing say\n");
    // Не раз, не два, а три раза в день!
    printf("Not once, or twice, but three times a day!\n");
    jerk();                                // Билл - безобразник
    // Он оскорбляет мою жену, моего кота, мою мать
    printf("He insulted my wife, my cat, my mother\n");
    // Он раздражает и морочит голову, как никто другой!
    printf("He irritates and grates, like no other!\n");
    jerk();                                // Билл - безобразник
    // Он хихикает, этот толстяк
    printf("He chuckles it off, his big belly a-heavin'\n");
    // Но он не будет смеяться, когда я доберусь до него!
    printf("But he won't be laughing when I get even!\n");
    jerk();                                // Билл - безобразник
    return(0);
}

/* Это - функция jerk() */
jerk()
{
    printf("Bill is a jerk\n");            // Билл - безобразник
}
```

Напечатайте исходный текст BIGJERK2.C в вашем редакторе. Обратите особое внимание на форматирование. Сохраните файл на диске под именем BIGJERK2.C. Скомпилируйте и выполните программу. Распечатка результатов этой программы практически такая же, как и распечатка результатов первой программы BIGJERK.

- ✓ Некоторые компиляторы в зависимости от настроек, связанных с сообщениями об ошибках, могут при компиляции BIGJERK2.C сгенерировать специальный тип сообщений об ошибках — предупреждения. Могут появиться следующие предупреждения (сообщения об ошибках): `no prototype` (нет прототипа); `Function should return a value` (Функция должна вернуть значение); `'jerk' undefined ('jerk' не определена)` или даже `no return value` (нет возвращаемого значения).

- ✓ Несмотря на эти ошибки, программа должна работать. Это просто “предупреждения” (а не грубые ошибки) о нарушении соглашений, обычно соблюдаемых в языке С. Это совсем не смертельные, неустранимые ошибки, в результате которых программа не смогла бы работать должным образом.
- ✓ Я объясню смысл этих предупреждений и расскажу, как исправить эти (незначительные) ошибки в следующих разделах. Держитесь крепко в седле.

Как функция `jerk()` работает в программе `BIGJERK2.C`

Функция работает подобно волшебной шкатулке. Она что-то делает. В программе `BIGJERK2.C` функция `jerk()` отображает строку текста на экране. Что-нибудь вроде `Bill is a jerk` (Билл — безобразник).

По программе `BIGJERK2.C` компьютер продвигается вперед, выполняя команды языка С как обычно, сверху донизу. Затем он сталкивается со строкой:

```
jerk();
```

Это — не ключевое слово языка С, и это — не функция, известная компилятору. Компьютер смотрит вокруг и находит функцию `jerk()`, определенную в вашем исходном тексте. Удовлетворенный, он перепрыгивает к определению и выполняет инструкции, записанные в этой функции. Когда компьютер достигает последней фигурной скобки в определении функции, он полагает, что он сделал все, что было записано в функции, и потому возвращается в основную программу — туда, где он был перед прыжком. В `BIGJERK2.C` такое случается три раза.

На рис. 20.1 действия компьютера иллюстрируются графически. Каждый раз, когда компьютер видит функцию `jerk()`, он выполняет команды из этой функции. Все происходит так, как если бы те инструкции были в коде на месте обращения к функции (как показано на рис. 20.1). Действительно, если вставить инструкции из функции на место обращения к ней, получим точную копию программы `BIGJERK1.C`.

```
void main()
{
    printf("He calls me on the phone with nothing say\n");
    • printf("Not once, or twice, but three times a day!\n");
    {
        jerk();
    }
    • printf("He insulted my wife, my cat, my mother\n");
    • printf("He irritates and grates, like no other!\n");
    {
        jerk();
    }
    • printf("He chuckles it off, his big belly a-heavin'\n");
    • printf("But he won't be laughing when I get even!\n");
    {
        jerk();
    }
}
•
•
```

Рис. 20.1. Выполнение функции в программе

- ✓ Компьютер все равно читает команды в исходном тексте сверху вниз в основной (главной) функции `main`. Однако когда компьютер видит другую функцию, такую как `jerk()`, он временно переходит к чтению ее кода, чтобы выполнить команды из этой функции. Затем он возвращается назад — туда, где был до этого.
- ✓ Имейте в виду, что не все функции столь же просты, как `jerk()`. Большинство функций содержит много строк программы. Было бы слишком сложно и неэкономно выписывать эти строки во всех тех местах программы, где можно обойтись обращением к функции.

Создание прототипов функций

Создание прототипов относят к тому виду искусства общения, в котором программист сообщает компилятору, что требуется функции. Это может показаться ненужным, но фактически именно наличие прототипа гарантирует, что функция используется должным образом; значит, прототип помогает вам следить за правильностью кода. Вероятно, вы сделаете вывод, что компилятор немного не доверяет вам. Но ведь вы также не очень доверяете ему, так что ваши отношения взаимны.

- ✓ *Proto* (прото) происходит от греческого слова “начало”.
- ✓ *Typing* (печатание) исходит от латинского слова, обозначающего “делать что-нибудь на клавиатуре”.

Основные проблемы создания прототипов

Чтобы убажить богов создания прототипов, вы должны сделать две вещи. Во-первых, сначала вы должны правильно описать саму функцию (например, функцию `jerk`). Измените строку

```
jerk()
```

в исходном тексте `BIGJERK2.C` так:

```
void jerk()
```

Эта строка говорит компилятору, что функция `jerk()` не возвращает никаких значений. Аналогичную строку можно записать для любой функции, тогда исчезнут предупреждения — сообщения об ошибках типа `function should return a value` (функция должна вернуть значение). (О функциях, возвращающих значения, рассказывается в главе 22 “Как на самом деле функционируют функции”; функции, которые не возвращают значение, имеют тип `void`. Так что пока я рассказываю о функциях, имеющих тип `void`.) Во-вторых, вы должны рассказать компилятору о функции (например, `jerk()`) в программе ранее, чем встретится обращение к этой функции. По существу, именно для этого и нужен прототип. Вы говорите компилятору: “Привет! В этой программе позже встретится функция (например, `jerk()`), и вот как ее правильно вызывать”. Чтобы сделать подобное уведомление, нужно поближе к началу программы написать строку, которая напоминает начало функции (например, `jerk()`), но заканчивается точкой с запятой:

```
void jerk(void);
```

Поместите эту строку между `#include <stdio.h>` и `int main()`, с чего начинается главная функция — функция `main()`. Начало программы будет выглядеть примерно так:

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

```
void jerk(void);
```

```
int main()
```

```
// главная функция
```


Строка `void jerk(void)`; и есть прототип. Прототип говорит, что компилятору позже в исходном тексте может встретиться функция `jerk()`. Кроме того, он говорит, что функция будет иметь тип `void`, и потому не будет возвращать никаких значений. Кроме того, функции `jerk()` не нужны никакие значения, которые могли бы передаваться в круглых скобках, — на это указывает слово `void` в ее круглых скобках. Везде `void`. Если вы не поняли, просто читайте дальше.

Внесите необходимые изменения в предыдущие команды с помощью редактора. Программа `BIGJERK2.C` во всем ее величии приведена ниже:

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

```
void jerk(void);
```

```
int main()                                // главная функция
{
    // Он звонит мне по телефону и ничего не говорит
    printf("He calls me on the phone with nothing say\n");
    // Не раз, не два, а три раза в день!
    printf("Not once, or twice, but three times a day!\n");
    jerk();                                // Билл - безобразник
    // Он оскорбляет мою жену, моего кота, мою мать
    printf("He insulted my wife, my cat, my mother\n");
    // Он раздражает и морочит голову, как никто другой!
    printf("He irritates and grates, like no other!\n");
    jerk();                                // Билл - безобразник
    // Он хихикает, этот толстяк
    printf("He chuckles it off, his big belly a-heavin'\n");
    // Но он не будет смеяться, когда я доберусь до него!
    printf("But he won't be laughing when I get even!\n");
    jerk();                                // Билл - безобразник
    return(0);
}
```

```
/* Это - функция jerk() */
void jerk()
{
    printf("Bill is a jerk\n");            // Билл - безобразник
}
```

Когда закончите, повторно сохраните `BIGJERK2.C` на диске. Перетранслируйте, и не беспокойтесь из-за различных ошибок-предупреждений.

- ✓ Прототип представляет собой (в основном) перефразировку функции, которая появится в программе позже.
- ✓ В прототипе необходимо указать, какая функция используется в программе, и описать, что именно должно быть между круглыми скобками.
- ✓ Прототип должен заканчиваться точкой с запятой. Это очень важно.
- ✓ Обычно я копирую первую строку функции и вставляю ее поближе к началу программы, а затем добавляю точку с запятой. Например, в `BIGJERK2.C` я скопировал строку `void jerk()` (начало функции `jerk()`) и вставил ее в начало исходного текста. Кроме того, приходится, конечно, добавлять необходимые слова `void` и точку с запятой.





- ✓ Нет, для главной функции `main()` прототип не нужен. Компилятор ожидает ее и знает все о ней (ну, может, почти все...).
- ✓ Требование наличия прототипов было добавлено к языку С уже после его создания. Вы можете столкнуться со старыми файлами исходных текстов на С, в которых прототипы отсутствуют совсем. В те далекие дни, когда были написаны такие программы (примерно до 1990 года), о прототипах программисты даже понятия не имели.

Трусливый способ избежать проблем создания прототипов

Только наиболее крутые гуру языка С знают об этой уловке — так что никому не говорите, что вы узнали об этом из книги для “чайников”!

Признаюсь, прототипы вносят некоторый беспорядок. Зачем повторять определение функции в начале программы, когда ее полное описание, очевидно, находится там, где записан ее код, и из этого описания компилятор все равно извлекает всю необходимую ему информацию? Не кажется ли вам прототип избыточным, а? Многие программисты думают, что прототипы избыточны, и по этой причине они кодируют свои программы снизу вверх. Вот так, как показано в приведенной ниже реализации программы BIGJERK:

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

```
/* функция jerk() */

void jerk(void)
{
    printf("Bill is a jerk\n");           // Билл - безобразник
}

/* Программа начинается здесь */

int main()                               // главная функция
{
    // Он звонит мне по телефону и ничего не говорит
    printf("He calls me on the phone with nothing say\n");
    // Не раз, не два, а три раза в день!
    printf("Not once, or twice, but three times a day!\n");
    jerk();                               // Билл - безобразник
    // Он оскорбляет мою жену, моего кота, мою мать
    printf("He insulted my wife, my cat, my mother\n");
    // Он раздражает и морочит голову, как никто другой!
    printf("He irritates and grates, like no other!\n");
    jerk();                               // Билл - безобразник
    // Он хихикает, этот толстяк
    printf("He chuckles it off, his big belly a-heavin'\n");
    // Но он не будет смеяться, когда я доберусь до него!
    printf("But he won't be laughing when I get even!\n");
    jerk();                               // Билл - безобразник
    return(0);
}
```

Отредактируйте исходный текст BIGJERK2.C. Для этого внесите изменения в предыдущую программу. Основное изменение заключается в том, что нужно удалить прототип `jerk()` и заменить его самой функцией `jerk()`. Обратите внимание, что функция `jerk()` определена как `void jerk(void)`, точно так же как прототип, но все-таки это сама функция, а не прототип.

Сохраните измененный исходный код на диске под именем BIGJERK3.C. Скомпилируйте и выполните. Вывод будет тот же самый, но, закодировав функцию снизу вверх, вы полностью исключили возможность сделать ошибки при создании прототипа.



- ✓ Программа все равно начинает выполняться с главной функции `main()`, несмотря на наличие любых других функций до главной. Да, компилятор-то, оказывается, хитер.
- ✓ Вы не обязаны кодировать ваши программы таким способом. Ведь программисты редко знают заранее, какие функции им понадобятся в дальнейшем, так что большинство программистов начинает программировать функции так, как это было сделано ранее в этой главе. Только после того, как функция написана, ее можно “вырезать и вставить” (командами Cut и Paste) в начало файла.
- ✓ Не ставьте точку с запятой после объявления функции, если вы сначала перечисляете свои функции! Если вы поставите лишнюю точку с запятой, то получите одну из тех противных ошибок синтаксического анализа (parse errors).
- ✓ Если исходный текст содержит несколько (больше одной) функций, важен порядок, в котором они перечисляются; вы не можете использовать функцию в исходном тексте, если она не была объявлена ранее или для нее не было создано прототипа. Если в исходном тексте есть несколько функций, упорядочивайте их так, чтобы вызываемая функция предшествовала вызывающей. Иными словами, если одна функция вызывает другую, то эта вторая функция должна быть описана ранее. В противном случае вы рискуете сделать ошибки при создании прототипов.

Функции — от А до Я

Язык C позволяет в исходном тексте использовать столько функций, сколько вы хотите. В действительности нет никакого ограничения, хотя большинство программистов любит сохранять разумный размер исходного текста, т.е. разумный размер текстовых файлов.

- ✓ Что такое “разумный размер”? Это зависит от многих обстоятельств.
- ✓ Чем больше файл исходного текста, тем дольше приходится его компилировать.
- ✓ Часто удобно вынести функции в их собственные, отдельные файлы исходного текста. Это не только помогает в отладке, но и упрощает перетрансляцию больших файлов.

Формат функции

Вот типичный формат функции:

тип имя(параметры)

Здесь *тип* указывает компилятору, возвращает ли функция значение. Если *тип* есть `void`, функция не возвращает никакого значения. (Это просто набор инструкций.) В противном случае *тип* указывает тип возвращаемого функцией значения. Это может быть `char`, `int`, `float` или любой из стандартных типов, допустимых в объявлении переменной в языке C.

Здесь *имя* — название (имя) функции. Оно должно быть уникальным названием (именем), и не может совпадать с любым из ключевых слов или названием (именем) другой библиотеч-

ной функции языка C, вроде `printf()` или `atoi()`. (Подробнее о названиях (именах) рассказывается в следующем разделе.) Круглые скобки после названия (имени) функции обязательны, поскольку они являются отличительным признаком всех функций языка C. В круглых скобках, если необходимо, указываются *параметры* — они определяют значение (или значения), передаваемые функции для вычислений, обработки или преобразований. Подробнее об этом рассказывается в главе 22 “Как на самом деле функционируют функции”. Если никаких параметров нет, круглые скобки можно оставить пустыми, либо вместо них можно указать слово `void`.

Инструкции, принадлежащие функции, необходимо заключить в фигурные скобки. Эти инструкции — команды, которые выполняют то, что должна сделать функция. Поэтому более полный формат функции выглядит так:

```
тип имя(параметры)
{
    инструкция(и);
    /* следующие инструкции */
}
```

Прототип функции должен быть создан до обращения к ней. Впрочем, создания прототипа можно избежать, если в исходном тексте поместить полное описание функции до обращения к ней. Ну, а при необходимости несложно создать и прототип. Он располагается в начале исходного текста и напоминает заголовок из объявления функции. Например:

```
тип имя(параметры);
```

Эта строка вместе с точкой с запятой и есть прототип функции. Если есть прототип функции, к этой функции можно обращаться в программе после прототипа. Создание прототипа требует только копирования (команда `Copy`) и вставки (команда `Paste`), но не забудьте, что в конце прототипа требуется также точка с запятой. (Если забудете, ваш компилятор может чрезвычайно мягко напомнить о ней серией сообщений об ошибках.)

- ✓ Запись функции называют определением функции. Называют ее также объявлением функции или созданием функции. (Официальный термин — определение функции.)
- ✓ Правила именования функций рассматриваются в следующем разделе.
- ✓ В справочном руководстве по библиотеке языка C при перечислении функций используются прототипы. Например:

```
int atoi(const char *s);
```

Этот формат (прототип) содержит требования и объясняет, что получится в результате обращения к функции `atoi()` в программе. Здесь *тип* — `int`, а *параметр* — `const char *s`. Это значит, что параметром функции должна быть символьная строка-константа, причем в функции она обозначена через `s`. (Кроме того, в справочном руководстве по библиотеке языка C при перечислении функций отмечают, какие директивы `#include` требуются в начале вашего исходного текста, чтобы можно было использовать описываемую функцию. Например, чтобы можно было использовать функцию `atoi()`, в начале исходного текста требуется директива `#include <stdlib.h>`.)



Именованние функций, или как давать названия (имена) функциям

Функции подобны вашим детям, поэтому, ради Бога, не давайте им глупых имен! Вы можете давать вашим функциям любые имена, но имейте в виду следующее:

- ✓ В названиях (именах) функций используют, как правило, буквы (символы алфавита) и цифры. Почти все компиляторы настаивают, чтобы ваши функции начинались с буквы (в крайнем случае допускаются и некоторые другие символы, например символ подчеркивания). Прочитайте документацию к вашему компилятору, чтобы выяснить все вопросы.

- ✓ Пробелы в именах функций не допускаются. Вместо них используется символ подчеркивания. Например, то, что вы видите ниже, не является именем функции:

```
get the time() // получить время ()
```

Но зато вот это вполне допустимо:

```
get_the_time()
```

- ✓ Вы можете использовать символы на верхнем и нижнем регистрах в названиях ваших функций. Обычная тактика состоит в том, чтобы в имени функции ключевые символы печатать прописными буквами:

```
getTheTime()
```

- ✓ Большинство компиляторов различает символы на верхнем и нижнем регистрах, так что если вы используете оба регистра, убедитесь, что помните, как правильно печатать имя с учетом регистра. Например, если вы функцию назвали `getTheTime`, а напечатали `GetTheTime`, то вы можете получить сообщение об ошибке от компоновщика (что-нибудь вроде *the function was not found* — функция не была найдена).

- ✓ Старайтесь давать функциям короткие и описательные (информативные) имена. Функцию можно назвать `f()` (поскольку имя `f` допустимо), но все же такое имя несколько неопределенно — оно ничего не говорит о том, что делает данная функция. Это все равно, что ответить “ни о чем”, если кто-то спросит вас, о чем вы думаете.

- ✓ Некоторые компиляторы могут запретить начинать имя функции с символа подчеркивания. Это звучит причудливо, но следующее имя может быть запрещено:

```
_whatever()
```

- ✓ Не называйте ваши функции теми же самыми именами, которые уже присвоены другим функциям языка C или совпадают с ключевыми словами. Будьте уникальны!

- ✓ Для первой выполняемой функции в вашей программе зарезервировано имя `main()`. (Это главная функция, поскольку с нее начинается выполнение программы).





Переменные в функциях

В этой главе...

- Обозначение переменных в функциях
- Понятие локальных переменных
- Совместное использование одной переменной во всей программе
- Использование глобальных переменных

Каждая функция, создаваемая программистом, может использовать ее собственный, частный набор переменных. Это обязательное требование. Точно так же как и в главной функции `main()`, в других функциях нужны целочисленные или символьные переменные, которые помогают функции выполнять ее задачу. Конечно, с концепцией использования переменных в функциях связано несколько причудливых правил.

В этой главе будут введены несколько странных понятий и концепций, связанных с использованием переменных в функциях. Эти переменные отличны от всех других, использующихся в программе. Они уникальны. Пожалуйста, не пытайтесь их уничтожить.

Сброс бомб в программе BOMBER для БОМБАРДИРОВЩИКА!

Функция `dropBomb()` в программе `BOMBER.C` использует ее собственную, частную переменную `x` в цикле `for`, чтобы моделировать сброс бомбы. Эта функция может использоваться в захватывающей разрабатываемой вами компьютерной игре, хотя вы, вероятно, захотите использовать сложную, впечатляющую графику, а не черно-белый экран пульта, как в данной версии:

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

```
void dropBomb(void); /* прототип */
```

```
int main()                                // главная функция
{
    // Нажмите <Enter>, чтобы сбросить бомбу
    printf("Press Enter to drop the bomb:");
    getchar();
    dropBomb();
    printf("Yikes!\n");
    return(0);
}
```

```
void dropBomb()
{
```

```

int x;

for(x=20;x>1;x--)
{
    puts("          *");           // отобразить ("*");
}
puts("          BOOM!");           // отобразить (" БУМ! ");
}

```

Напечатайте исходный текст так, как он здесь представлен. В функции `puts()`, в строке `puts(" *");` — 10 пробелов перед звездочкой. В строке `puts(" BOOM!");` — 8 пробелов перед `BOOM!` (БУМ!).

Сохраните файл на диске под именем `BOMBER.C`. Скомпилируйте и выполните. Нажмите `<Enter>`, чтобы сбросить бомбу:

```

*
*
*
*
*

```

И так далее....

```

*
*
*
*

```

BOOM!

Yikes!

Да, это происходит слишком быстро, чтобы пользователь успел испугаться и возникло нервное напряжение, характерное для настоящей видеоигры, но суть здесь не в том, чтобы сбросить бомбы; главное здесь то, что переменная `x` используется в функции `dropBomb()`. Это работает просто прекрасно. Никаких ошибок. Ничего нового. Именно так переменные используются в функциях.

- ✓ Обратите внимание на то, как в функции `dropBomb()` объявлена переменная `x`:

```
int x;
```

Все делается точно так же, как и в главной функции `main()`: переменные объявляются в начале функции. (Объявление переменных подробнее описано в главе 8 “Переменные в языке C”.)

- ✓ Функция `dropBomb()` имеет тип `void`, потому что она не возвращает никакого значения. Сам тип функции не имеет никакого отношения к значениям, используемым в функции.

Будет ли бомбить программа `BOMBER.C` с двумя переменными?

Давайте теперь с помощью редактора изменим исходный текст `BOMBER.C`. Нужно изменить главную функцию `main()` вот так:

```

int main()                               // главная функция
{
    char x;

    // Нажмите <Enter>, чтобы сбросить бомбу
    printf("Press Enter to drop the bomb:");
    x=getchar();
    dropBomb();
}

```



```
// Код клавиши %d, чтобы сбросить бомбу
printf("Key code %d used to drop bomb.\n",x);
return(0);
}
```

Мы создали еще одну переменную *x* в главной функции *main()*. Эта переменная используется независимо от переменной *x* в функции *dropBomb()*. Чтобы доказать это, сохраните измененный файл на диске, а затем скомпилируйте его, и выполните получившуюся программу.

Вывод этой измененной версии программы будет практически тот же самый; отличие состоит лишь в том, что отображается код клавиши <Enter> (можно сказать, что это "символ", который генерируется клавиатурой) — 10 или 13, в зависимости от вашего компьютера.



- ✓ Обратите внимание, что компьютер использует две переменные *x* и не путается при этом? Каждая переменная сохраняет свое значение.
- ✓ Названия (имена) переменных в различных функциях могут совпадать.
- ✓ В программе, например, может быть десяток различных функций, в каждой из которых используется переменная с одним тем же самым именем. Это вполне законно. Все эти переменные независимы друг от друга, потому что каждая из них вложена в свою собственную функцию.
- ✓ Переменная *x* используется не просто в двух различных функциях. Фактически это две различные, независимые друг от друга переменные, притом типы этих двух переменных различны: символ и целое число. Сверхъестественно!

Важное добавление: некоторая психологическая напряженность

Признайтесь: ни в одной игровой программе бомбы не сбрасывались так быстро, как в BOMBER.C. Нужна пауза, чтобы создать немного *напряженности* и усилить волнение игрока во время падения бомбы. Почему бы не придумать собственную функцию *delay()*, чтобы создать такую психологическую напряженность? Вот модифицированный исходный текст:

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

```
// СЧЕТЧИК
```

```
#define COUNT 20000000 /* 20,000,000 */
```

```
void dropBomb(void); /* прототип */
```

```
void delay(void);
```

```
int main()
```

```
// главная функция
```

```
{
```

```
    char x;
```

```
    // Нажмите <Enter>, чтобы сбросить бомбу
```

```
    printf("Press Enter to drop the bomb:");
```

```
    x=getchar();
```

```
    dropBomb();
```

```
    // Код клавиши %d, чтобы сбросить бомбу
```

```
    printf("Key code %d used to drop bomb.\n",x);
```

```
    return(0);
```

```
}
```

```
void dropBomb()
```

```
{
```

```

int x;

for(x=20;x>1;x--)
{
    puts("      *");           // отобразить ("*");
    delay();                  // задержка
}
puts("      BOOM!");          // отобразить (" БУМ!");
}

void delay()
{
    long int x;

    for(x=0;x<COUNT;x++)
        ;
}

```

Обратите внимание на изменения! Вот они:

```

// СЧЕТЧИК
#define COUNT 20000000 /* 20 000 000 */

```

Здесь объявлена константа COUNT (СЧЕТЧИК). Она равна 20 миллионам. Это — значение задержки. (Если задержка слишком велика, уменьшите ее; если задержка слишком мала, сделайте значение большим.)

Затем следует прототип функции delay():

```
void delay(void);
```

Вызов функции delay() добавляется в функцию dropBomb() сразу после вызова puts(), который отображает “бомбу”:

```
delay();                      // задержка
```

Наконец, следует код функции delay(). Длинное целое число хранится в переменной x — отличной от любой другой переменной x, используемой в любой другой функции. Это целое число используется в цикле for (для), чтобы создать задержку.

Перепроверьте исходный текст. Сохраните его на диске. Скомпилируйте и выполните. На сей раз психологическое напряжение возрастает по мере того, как бомба медленно падает на землю, а затем... БУМ!



- ✓ Строка, в которой создается константа (эта строка начинается с #define), не заканчивается точкой с запятой! В главе 8 “Переменные в языке C” объясняется, почему.
- ✓ Вы можете корректировать задержку, изменяя постоянную COUNT (СЧЕТЧИК). Обратите внимание, насколько это проще — не нужно охотиться по всей программе, чтобы найти сам цикл задержки и значение задержки в нем. Это одно из главных преимуществ использования переменной-постоянной.
- ✓ Переменные с теми же самыми названиями (именами) в различных функциях независимы друг от друга.
- ✓ Что здесь самое важное? Я думаю, что главный урок состоит в том, что вы не должны придумывать новые имена переменных в каждой функции. Например, если вы любите использовать в цикле for переменную i, вы можете использовать i во всех ваших функциях. Только объявляйте их, а о совпадении имен не волнуйтесь.

Глобальные переменные: совместное использование и всеобщая любовь

Иногда несколько функций должны совместно использовать одну и ту же переменную. Большинство игр, например, сохраняет счет в переменной, которая доступна для множества функций. Она должна быть доступна, например, для функции, которая отображает счет на экране; для функции, которая увеличивает значение счета; для функции, которая уменьшает значение; а также для функций, которые сохраняют счет на диске. Все эти функции должны обращаться к одной и той же переменной. Чтобы это сделать, нужно создать глобальную переменную.

Глобальная переменная — это переменная, которую может использовать любая функция программы. Главная функция `main()` также может использовать ее. Таким образом, глобальную переменную могут использовать все функции. Они могут изменить (заменить) ее значение, проверить его, записать значение переменной на диск и т.д. Никаких проблем.

Противоположность глобальной переменной — локальная переменная (с локальными переменными вы познакомитесь далее в этой книге). Локальная переменная существует в только одной функции — подобно переменной `x` в программе `BOMBER.C`. Переменная `x` — локальная переменная, уникальная для каждой функции, в которой она создается; каждая локальная переменная игнорируется другими функциями программы.

- ✓ Глобальная переменная доступна всем функциям программы.
- ✓ Локальная переменная доступна только функции, в которой она создана.
- ✓ Глобальные переменные могут использоваться в любой функции, причем повторно объявлять их не надо. Например, если в программе уже есть объявление глобальной целой переменной `score` (счет), то в каждой функции, которая использует эту переменную, повторять объявление `int score`; не нужно. Переменная была уже объявлена и может использоваться в любой функции.
- ✓ Поскольку глобальные переменные доступны во всей программе, важно удачно назвать их. Ведь после того, как вы объявите глобальную переменную, например, `x`, ни в какой другой функции не удастся объявить новую переменную `x` так, чтобы глобальная переменная `x` была доступна в этой функции. В случае совпадения имен компилятор может выдать даже предупреждение.



Создание глобальной переменной

Глобальные переменные отличаются от локальных переменных в двух отношениях. Во-первых, поскольку они — глобальные переменные, любая функция в программе может использовать их. Во-вторых, они объявляются вне любой функции: там, где ни одна функция не может завладеть ими монопольно. В пустоте. Среди хаоса. Странно, но истинно.

Например, вот так:

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

```
int score;                                // счет
```

```
int main()                                // главная функция
{
```

И т.д. ...

Представьте себе, что этот исходный текст — начало некоторой огромной программы, подробности которой сейчас не важны. Обратите внимание, как объявлена переменная `score`. Само объявление сделано вне любой функции. По традиции оно расположено как раз перед главной функцией `main()` и после строк, начинающихся со знака фунта (и после всех прототипов, если они есть). Именно благодаря такому расположению объявления переменные становятся глобальными.

Если требуется несколько глобальных переменных, они создаются на том же самом месте, т.е. непосредственно перед главной функцией `main()`. Само объявление их ничем не отличается от обычного; единственное различие состоит только в том, что оно расположено вне любой функции.

- ✓ Глобальные переменные объявляются вне любой функции. Это обычно делается прямо перед главной функцией `main()`.
- ✓ Все, что вы знаете о создании переменных, кроме того, что связано с расположением объявления вне функции, относится и к созданию глобальных переменных: вы должны определить тип переменной (например, `int`, `char` или `float`), указать название (имя) переменной и после него поставить точку с запятой.
- ✓ Вы можете также сразу объявить несколько глобальных переменных:

```
int score,tanks,ammo;           // счет, танки, боеприпасы;
```
- ✓ Кроме того, при необходимости глобальным переменным можно присвоить значения:

```
char prompt[]="What?";
```

Пример глобальной переменной в реальной, действующей программе

Удовольствия ради давайте снова обратимся к исходному тексту `BOMBER.C`. В приведенной ниже окончательной модификации добавлен код, который сохраняет текущее общее количество людей, уничтоженных бомбами. Это общее количество сохраняется в глобальной переменной `deaths` (смертельные случаи), определенной в самом начале программы. Вот окончательный исходный текст, со всеми изменениями, анализ которых мы выполним несколько позже:

```
#include <stdio.h>

// СЧЕТЧИК
#define COUNT 20000000 /* 20 000 000 */

void dropBomb(void); /* прототип */
void delay(void);

int deaths; /* глобальная переменная */ // смертельные случаи;

int main()                                     // главная функция
{
    char x;

    deaths=0;
    for(;;)
    {
        // Нажмите <~>, чтобы закончить
        printf("Press ~ then Enter to quit\n");
```

```
// Нажмите <Enter>, чтобы сбросить бомбу
printf("Press Enter to drop the bomb:");
x=getchar();

fflush(stdin); /* очистка буфера ввода */
if(x=='-')      /* если (x == '-')
{
    break;      // прервать
}
dropBomb();
// (" %d людей уничтожено!\n ", смертельные случаи)
printf("%d people killed!\n",deaths);
}
return(0);
}

void dropBomb()
{
    int x;
    for(x=20;x>1;x--)
    {
        puts("          *");      // отобразить ("*");
        delay();                  // задержка
    }
    puts(" BOOM!");              // отобразить (" БУМ! ");
    deaths+=1500;                 // смертельные случаи += 1500;
}

void delay()
{
    long int x;
    for(x=0;x<COUNT;x++)
    ;
}
```

Откройте исходный текст BOMBER.C в вашем редакторе. Сделайте необходимые изменения так, чтобы ваш код имел приведенный выше вид. Вот некоторые подробности редактирования.

Во-первых, глобальная переменная `deaths` объявляется прямо перед главной функцией `main()`:

```
int deaths; /* глобальная переменная */
```

Во-вторых, главная функция `main()` переписана, теперь она зациклена, так как содержит бесконечный цикл (см. код). Цикл позволяет сбрасывать бомбы много раз; нажимая клавишу <-> (тильда), игрок заканчивает цикл.

В-третьих, глобальная переменная `deaths` увеличивается в функции `dropBomb()` простым применением операции сложения в этой функции:

```
deaths+=1500;      // смертельные случаи += 1500;
```

Перепроверьте введенный исходный текст! Затем сохраните BOMBER.C на диске еще раз (хотя вы не закончите эту программу, пока не прочитаете главу 22 "Как на самом деле функционируют функции").

Скомпилируйте и выполните программу.

Программа в ходе выполнения фактически сохраняет число мертвых в глобальной переменной `deaths`. Переменная изменяется (увеличивается) в функции `dropBomb()` и отображается в главной функции `main()`. Обе функции совместно используют эту глобальную переменную.



- ✓ Обратили внимание, перед чем объявлена глобальная переменная? Сначала обычно следуют строки `#include`, затем — прототипы, а после них — глобальные переменные.
- ✓ Нет, в этой игре ничего интересного нет. Противник не ведет активных действий.
- ✓ Более подробно бесконечный цикл `for` описан в главе 25 “Математическое безумие!”.
- ✓ Функция `fflush()` нужна для очистки буфера от дополнительных символов, которые могли бы быть прочитаны функцией `getchar()`. Подробности содержатся в главе 13 “Сравнение символов с помощью ключевого слова `if`”.
- ✓ В семействе операционных систем Unix убедитесь, что использовали `fpurge(stdin)`, а не `fflush(stdin)`. Обратитесь к главе 13 “Сравнение символов с помощью ключевого слова `if`”.

Как на самом деле функционируют функции

В этой главе...

- Передача значения функции
- Передача нескольких значений функции
- Использование ключевого слова `return`
- Понятие главной функции `main()`
- Более плотная запись кода

Ф

ункция походит на машину. Хотя пустые (`void`) функции — лентяи, о которых вы, вероятно, читали в более ранних главах, все равно они — вполне законные функции: реальное значение функции состоит в том, что она делает. Я подразумеваю, что функции должны перерабатывать что-нибудь и выдавать результат. Нечто вроде мясорубки. Функции должны функционировать.

Эта глава объясняет, как использовать функции для управления информацией или обработки (переработки) ее. Чтобы выполнить такие действия, нужно суметь передать значение функции и заставить функцию вернуть значение. Данная глава объясняет, как все это делается.

Передача значения функции

Вообще говоря, есть четыре типа функций:

- ✓ **Функции, которые делают все сами и им не требуется никакая дополнительная информация.** Такие функции описаны в предыдущих главах. Каждая такая функция немного замкнута, но часто необходима и, несомненно, является функцией не в меньшей степени, чем любая другая функция.
- ✓ **Функции, которые берут что-нибудь на входе и используют это тем или иным способом.** Этим функциям передаются значения — константы или переменные, — которые они пережевывают и затем делают нечто полезное из полученных ими значений.
- ✓ **Функции, которые берут что-нибудь на входе и возвращают результат.** Эти функции получают кое-что и возвращают кое-что назад (это называется возвращением результата). Например, функция, которая вычисляет ваш вес исходя из вашего размера обуви, принимает на входе ваш размер обуви и возвращает ваш вес — ввод и вывод.
- ✓ **Функции, которые только возвращают результат.** Эти функции генерируют значение или строку, возвращая его программе. К этому типу относится, например,

функция, которая может выяснить, где находится корабль “Энтерпрайз” в Империи Клингон. Вы вызываете функцию `whereEnt()`, а она возвращает некоторые галактические координаты.

Любая функция может быть отнесена к одной из этих категорий. Выбор категории при создании функции полностью зависит от того, что вы хотите, чтобы функция сделала. Когда вы будете это знать, вы создадите нужную функцию.

Как передать значение функции

Передать значение функции столь же просто, как пролезть через игольное ушко. Нужно только действовать последовательно и выполнить следующие шаги.

1. **Определите, какое значение вы собираетесь передавать функции.** Это может быть постоянное значение, число или строка, или даже переменная (поименованная, конечно, по всем правилам языка C). Как бы то ни было, вы должны объявить переменную, т.е. правильно указать тип значения, чтобы функция знала точно, какое значение она получает. В качестве типа, например, можно указать `int`, `char` или `float`.
2. **Объявить значение можно так же, как переменную, но сделать это нужно в круглых скобках функции.** Предположим, что ваша функция принимает целочисленное значение. Это значение нужно объявить так же, как объявляется переменная, которую функция будет использовать в дальнейшем. Иными словами, объявление передаваемого значения подобно объявлению любой переменной в языке C за тем исключением, что это объявление нужно сделать внутри круглых скобок функции после имени функции:

```
void jerk(int repeat)
```

Однако после объявления переменной в этом случае точка с запятой не ставится! В предыдущем примере объявлена целочисленная переменная `repeat` (повторение), что означает, что функция `jerk()` требует целочисленного значения. Внутри функции обращение к этому значению выполняется с помощью переменной `repeat`.

3. **Тем или иным способом используйте значение в вашей функции.** Компилятор не любит, если переменная объявлена, но данная переменная нигде больше не используется. (Это расход памяти.) Сообщение об ошибках выглядит примерно так: `jerk is passed a value that is not used (jerk передался значение, которое не используется)` или так: `Parameter 'repeat' is never used in function jerk` (Параметр 'repeat' нигде не используется в функции `jerk`). Это сообщение об ошибке — всего лишь предупреждение. И компилятор иногда может даже не обнаружить такую ошибку. Но лучше всего придерживаться правила: используйте все объявленные вами переменные!
4. **Правильно запишите прототип функции.** Это нужно сделать для того, чтобы избежать серии предупреждений-ошибок. Мой совет: выделите строку, с которой начинается ваша функция; отметьте ее как блок текста. Затем вставьте ее в начало программы, чуть выше главной функции `main()`. После вставки скопированного блока добавьте точку с запятой. Вот пример:

```
void jerk(int repeat);
```

Никаких проблем.



5. Не забывайте передавать соответствующие значения, когда вы вызываете функцию. Поскольку функция обязана принимать значения, вы обязательно должны их передавать. Не должно быть никаких пустых круглых скобок! Вы должны заполнить их, и заполнить их значениями надлежащих типов (например, целым числом, символом, числом с плавающей точкой). Только в этом случае функция сможет должным образом выполнить возложенную на нее задачу.



- ✓ Параметр иногда называется аргументом. Этот термин напоминает, что по некоторым темам программирования на языке C вы можете вступать в дискуссии.
- ✓ Название (имя), данное параметру функции (т.е. передаваемой переменной, аргументу, значению и т.д.), используется в определении и прототипе функции, а также в самом коде функции.
- ✓ Параметр функции можно рассматривать как локальную переменную. Да, она определяется в прототипе. Да, она появляется на первой строке. Но внутри функции это всего лишь локальная переменная.
- ✓ Между прочим, имя переменной, используемое в функции, должно совпадать с именем переменной, определенной в круглых скобках функции. (Подробнее об этом читайте далее.)
- ✓ Передача значения функции и возврат значения функцией — это не то же самое, что использование глобальной переменной. Хотя функция может использовать глобальные переменные, значения, возвращаемые функцией, не являются глобальными переменными. (Подробнее о глобальных переменных рассказывается в главе 21 “Переменные в функциях”).

Пример (в самый раз!)

Вслепую напечатайте следующую программу, являющуюся модификацией серии программ BIGJERK.C, с которыми вы познакомились в главе 20 “Создание первой функции”:

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

```
void jerk(int repeat);  
int main()                                // главная функция  
{  
    // Он звонит мне по телефону и ничего не говорит  
    printf("He calls me on the phone with nothing say\n");  
    // Не раз, не два, а три раза в день!  
    printf("Not once, or twice, but three times a day!\n");  
    jerk(1);  
    // Он оскорбляет мою жену, моего кота, мою мать  
    printf("He insulted my wife, my cat, my mother\n");  
    // Он раздражает и морочит голову, как никто другой!  
    printf("He irritates and grates, like no other!\n");  
    jerk(2);  
    // Он хихикает, этот толстяк  
    printf("He chuckles it off, his big belly a-heavin'\n");  
    // Но он не будет смеяться, когда я доберусь до него!  
    printf("But he won't be laughing when I get even!\n");  
    jerk(3);  
    return(0);  
}
```

```

/* Функция jerk() повторяет рефрен количество раз,
   определяемое значением переменной repeat (повторение) */
void jerk(int repeat)
{
    int i;

    for(i=0;i<repeat;i++)
        printf("Bill is a jerk\n");    // Билл - безобразник!
}

```

Вы можете отредактировать исходный текст из файла BIGJERK2.C, но сохраните полученный файл на диске под именем BIGJERK4.C. В этом файле сделаны некоторые изменения, главным образом в функции `jerk()` и инструкциях вызова этой функции. Не пропустите чего-нибудь, иначе вы получите несколько противных сообщений об ошибках.

Скомпилируйте и выполните.

Вывод программы теперь выглядит вот так:

```

He calls me on the phone with nothing say
Not once, or twice, but three times a day!
Bill is a jerk
He insulted my wife, my cat, my mother
He irritates and grates, like no other!
Bill is a jerk
Bill is a jerk
He chuckles it off, his big belly a-heavin'
But he won't be laughing when I get even!
Bill is a jerk
Bill is a jerk
Bill is a jerk

```

(Он звонит мне по телефону и ничего не говорит

Не раз, не два, а три раза в день!

Билл — безобразник

Он оскорбляет мою жену, моего кота, мою мать

Он раздражает и морочит голову, как никто другой!

Билл — безобразник

Билл — безобразник

Он хихикает, этот толстяк

Но он не будет смеяться, когда я доберусь до него!

Билл — безобразник

Билл — безобразник

Билл — безобразник)

Мы изменили функцию `jerk()`! Она теперь может отобразить унылый рефрен любое указанное число раз. Удивительно! Вот что функция может сделать для поэзии!

Подробно о том, как работает эта программа, рассказывается в остальной части этой главы. Следующие замечания могут предвосхитить несколько ключевых вопросов.

- ✓ Обратите внимание, как функция `jerk()` определена с помощью прототипа:

```
void jerk(int repeat);
```

Эта строка сообщает компилятору, что функция `jerk()` жаждет целочисленного значения, которое называется `repeat` (повторение).

- ✓ Новая функция `jerk()` повторяет фразу `Bill is a jerk` (Билл — безобразник) любое указанное количество раз. Например:

```
jerk(500);
```



Эта инструкция вызывает функцию `jerk()`, которая повторяет сообщение 500 раз.

- ✓ Знатоки языка C употребляют глагол “передать” для того, чтобы обозначить посылку значения переменной функции. Так что вы передаете значение 3 функции `jerk()` в следующей инструкции:

```
jerk(3);
```

- ✓ Значение, передаваемое функции, называется параметром. Например, можно сказать, что функция `jerk()` имеет один параметр — целочисленную переменную (в данном случае обозначенную числом).
- ✓ Если вы забудете передать значение функции `jerk()`, вы получите сообщение об ошибке `Too few parameters` (Слишком мало параметров). Именно по этой причине нужно указывать прототип функции. Если этого не сделать и использовать `jerk()` в качестве вызова функции, при выполнении программы неизбежно возникнет ошибка.
- ✓ Переменная `repeat` (повторение) — это не глобальная переменная. Это просто значение, которое передается функции `jerk()`, и именно его после передачи использует эта функция, чтобы сделать то, что задумано программистом.
- ✓ Обратите внимание, что в `jerk()` есть также своя собственная переменная `i`. Ничего нового в этом для вас нет.

Как избежать беспорядка в переменных (обязательное чтение)



Вы не обязаны вызывать функцию, используя то же самое имя переменной, которое используется в функции. Не вздумайте перечитывать это предложение дважды. Это — несколько путаная концепция, так что читайте дальше.

Предположим, что из главной функции `main()`, где используется переменная, названная `count` (счет), вы хотите передать значение функции `jerk()`. Вот как это делается:

```
jerk(count);
```

Эта строка велит компилятору вызвать функцию `jerk()` и передать ей значение переменной `count`. Поскольку `count` — целая переменная, все это работает просто прекрасно. Но имейте в виду, что передается значение переменной. Переменная называется `count`. Но `count` — только название (имя), используемое в некоторой функции. Оно не важно! Важно только значение.

В функции `jerk()` значение обозначается переменной с именем `repeat`. Вот ведь как была объявлена функция `jerk()`:

```
void jerk(int repeat)
```

Независимо от передаваемого значения, оно в функции всегда обозначается как `repeat`.

- ✓ Я объясняю эту концепцию, потому что она немного туманна. Вы можете вызвать любую функцию, используя любое имя переменной. Только в вызываемой функции используется то имя переменной, которое было указано при определении функции.



✓ Данный материал можно назвать сложным именно из-за имен переменных, используемых в функции. Несомненно, эта тема освещается также в вашем руководстве по языку C. Рассмотрим следующий пример функции:

```
int putchar(int c);
```

Эта строка указывает, что в функции `putchar()` используется целочисленное значение, которое в самой функции обозначается через `c`. Однако можно вызвать эту функцию, используя любое имя переменной или постоянное значение. Ведь имя не важно. Важно лишь то, что это должна быть целочисленная переменная.

Передача нескольких значений функции

На самом деле функции можно передать не одно значение, а несколько. Нужно лишь заключить передаваемые значения в круглые скобки функции. В объявлении функции каждый параметр имеет тип и название (имя), причем после параметра (кроме последнего) следует запятая. Точка с запятой в конце не ставится.

Например:

```
void bloat(int calories, int weight, int fat)
```

В определении этой функции указано, что в ней используется три целочисленных значения: `calories` (калории), `weight` (вес) и `fat` (жир). Но не все эти переменные обязаны иметь один и тот же тип. Пример разных типов параметров мы имеем в следующем объявлении:

```
void jerk(int repeat, char c)
```

Здесь вы видите модификацию функции `jerk()`, в которой теперь используется два значения: целое число и символ. Эти значения в функции `jerk()` обозначаются через `repeat` (повторение) и `c`. Вот исходный текст с этой функцией:

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

```
void jerk(int repeat, char c);
```

```
// главная функция
int main()
{
    // Он звонит мне по телефону и ничего не говорит
    printf("He calls me on the phone with nothing say\n");
    // Не раз, не два, а три раза в день!
    printf("Not once, or twice, but three times a day!\n");
    jerk(1, '?');
    // Он оскорбляет мою жену, моего кота, мою мать
    printf("He insulted my wife, my cat, my mother\n");
    // Он раздражает и морочит голову, как никто другой!
    printf("He irritates and grates, like no other!\n");
    jerk(2, '?');
    // Он хихикает, этот толстяк
    printf("He chuckles it off, his big belly a-heavin'\n");
    // Но он не будет смеяться, когда я доберусь до него!
    printf("But he won't be laughing when I get even!\n");
    jerk(3, '!');
    return(0);
}
```

```
/* функция jerk() повторяет рефрен количество раз,
   определяемое значением переменной repeat (повторение) */
```

```
void jerk(int repeat, char c) {
    int i;

    for(i=0;i<repeat;i++)
        printf("Bill is a jerk%c\n",c);
}
```

Напечатайте предыдущий исходный текст. Вы можете использовать сходный текст BIGJERK4.C как ядро. Вы должны отредактировать (изменить) прототип функции `jerk()`; отредактируйте также вызовы функции `jerk()` (не забудьте и об одинарных кавычках); затем переопределите саму функцию `jerk()`; наконец, измените инструкцию `printf()` в функции так, чтобы она отображала символьную переменную `c`.

Сохраните файл на диске под именем BIGJERK5.C.

Скомпилируйте и выполните программу. Вывод выглядит почти так же, но вы увидите эффекты передачи односимвольной переменной функции `jerk()` — обратите внимание на то, как отображаются вопросительные и восклицательные знаки:

```
He calls me on the phone with nothing say
Not once, or twice, but three times a day!
Bill is a jerk?
He insulted my wife, my cat, my mother
He irritates and grates, like no other!
Bill is a jerk?
Bill is a jerk?
He chuckles it off, his big belly a-heavin'
But he won't be laughing when I get even!
Bill is a jerk!
Bill is a jerk!
Bill is a jerk!
```

(Он звонит мне по телефону и ничего не говорит

Не раз, не два, а три раза в день!

Билл — безобразник?

Он оскорбляет мою жену, моего кота, мою мать

Он раздражает и морочит голову, как никто другой!

Билл — безобразник?

Билл — безобразник?

Он хихикает, этот толстяк

Но он не будет смеяться, когда я доберусь до него!

Билл — безобразник!

Билл — безобразник!

Билл — безобразник!)



Еще один способ спорить с функциями — засыпать их аргументами

В этой книге вы уже познакомились с современным, удобным способом объявления переменных (параметров) в функции. Вот пример:

```
void jerk(int repeat, char c);
```

```
{
```

и так далее...

Однако вы можете также использовать старый формат:

```
void jerk(repeat, c)
int repeat;
char c;
{
```

и так далее...

Это объявление делает то же самое, но оно несколько более туманное, потому что сначала указано имя переменной и только затем ее объявление, причем объявление перенесено в следующую строку (или строки). Во всем остальном эти два объявления не отличаются.

Мой совет: придерживайтесь формата, используемого в этой книге, но не впадайте в панику, если вы увидите другой формат. В более старых справочниках по C приводится второй формат, и некоторые старые программисты, которые привыкли к старому C, могут придерживаться этого более старого формата. Остерегайтесь!

Функции, возвращающие значения

Некоторые функции, чтобы их можно было использовать должным образом, должны вернуть значение. Вы передаете ваш день рождения, и функция волшебным образом говорит, сколько вам лет, а затем компьютер хихикает над вами. Это пример возврата значения, и большинство функций ведут себя именно так.

Кое-что неприятное: правила

Чтобы возвращать значение, функция должна повиноваться следующим двум правилам.



Предупреждение! Приближаются правила.

- ✓ В определении функции должен быть указан ее тип. Например, `int`, `char` или `float` — у функции может быть точно тот же тип, что и у переменной. Укажите все что угодно, но не `void`.
- ✓ Функция должна вернуть значение.

Тип функции указывает тип значения, возвращаемого функцией. Например:

```
int birthday(int date);
```

В этой строке определена функция `birthday()`. Это целочисленная функция и возвращает она также целочисленное значение. (Ей также нужно передать целочисленный параметр — `date` (дата), который она использует как входные данные.) Следующая функция `nationalDebt()` возвращает национальный долг Соединенных Штатов как значение типа `double` (с удвоенной точностью):

```
double nationalDebt(void)
```

Ключевое слово `void` в круглых скобках означает, что функции не нужны никакие входные данные. Аналогично, когда функция не возвращает никакого результата, тип ее определяется как `void`:

```
void USGovernment(float tax_dollars)
```

Функции `USGovernment()` в качестве входных данных требуются очень большие числа, однако результата она не возвращает, поэтому данная функция имеет тип `void`. Просто запомнить.



- ✓ Любое значение, вычисленное функцией, возвращается с помощью ключевого слова `return`. Подробно это слово обсуждается далее в этой главе.
- ✓ Обратите внимание, что функции языка C, подобные `atoi()` и `getchar()`, возвращают значения. Эти функции должны быть перечислены в вашем справочном руководстве по библиотеке языка C, причем в нем используется тот же самый формат, принятый в данной книге:

```
int atoi(char *s)
char getchar(void)
```

Это означает, что функция `atoi()` возвращает целочисленное значение и что `getchar()` возвращает односимвольное значение.

- ✓ Есть еще одна причина создания прототипов функции: компилятор перепроверяет, что функция возвращает надлежащее значение и что в других частях программы используются типы вроде `int`, `float` или `char` так, как указано в определении функции.
- ✓ Чтобы вычислить национальный долг, нужны переменные типа `double`. Переменная типа `float`, хотя и может хранить числа, равные нескольким триллионам, имеет точность только 7 цифр. Переменные типа `double` имеют точность 15 цифр. Если бы долг был вычислен как число типа `float`, точность не превышала бы 100 000 долларов (совсем маленькое значение!).
- ✓ Хотя можно в определении функции указать тип `void`, переменную типа `void` объявить нельзя. Это была бы ошибка.
- ✓ Функции тип `void` не возвращают значений.
- ✓ Функция может вернуть только одно значение. В отличие от передачи значений функции, где функция может получить любое количество значений, вернуть функция может только одно значение. Я знаю, вы разочарованы, но это так...
- ✓ Впрочем, не совсем... Функция может вернуть несколько значений. Она делает это через чудо-указатели и структуры. Эти два дополнительных средства рассмотрены в книге *C All-in-One Desk Reference For Dummies*, выпущенной издательством "Wiley".

Наконец, компьютер вычисляет ваш IQ

Следующая программа вычисляет ваш показатель интеллекта IQ. Возможно. Но гораздо важнее то, что в этой программе используется функция, которая возвращает значение. Если вы прочитали несколько предыдущих глав, вы знакомы со следующими инструкциями языка C, используемыми для ввода данных с клавиатуры:

```
input=gets(); // ввод
x=atoi(input);
```

Функция `gets()` считывает текст с клавиатуры, а `atoi()` преобразует его в целочисленное значение. Ну, а все это в программе IQ. Сделает функция `getval()`, возвращая значение главной функции `main()`:

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

```
int getval(void);
```

```
int main() // главная функция
{
```

```

int age,weight,area;           // возраст, вес, область;
float iq;

// Программа вычисляет ваш IQ
printf("Program to calculate your IQ.\n");
printf("Enter your age:");      // Введите ваш возраст
age=getval();
printf("Enter your weight:");   // Введите ваш вес
weight=getval();
printf("Enter the your area code:"); // Введите код вашего города
area=getval();
iq=(age*weight)/area;           // iq = (возраст * вес) / область
// (*Компьютер считает, что ваш показатель интеллекта равен %f.\n", iq)
printf("This computer estimates your IQ to be %f.\n",iq);
return(0);
}

int getval(void)
{
    char input[20];
    int x;

    gets(input);
    x=atoi(input);             // ВВОД
    return(x);
}

```

Напечатайте исходный текст IQ.C с помощью вашего редактора. Новое здесь — функция `getval()`, которая возвращает значение, используя ключевое слово `return`. (Оно вам знакомо? Через секунду узнаете о нем больше!) Сохраните файл с исходным текстом на диске под именем IQ.C.

Скомпилируйте и выполните.

Вот как примерно выглядит вывод:

```

Enter your age:33
Enter your weight:175
Enter your area code:208
The computer estimates your IQ to be 27.000000.

```

(Введите ваш возраст:33

Введите ваш вес:175

Введите код вашего города:208

Компьютер считает, что ваш показатель интеллекта равен 27,000000.)

Конечно, я знаю, что мой показатель интеллекта так высок. Я нисколько не хвастаюсь. В конце концов, это только результат вычислений.

- ✓ Эта формула показывает, что только старые, тучные люди, живущие в городах с большими кодами, имеют шанс стать академиками.
- ✓ Данная программа имеет некоторые недостатки. Например, значение показателя интеллекта IQ, вычисляемое этой программой, является числом с плавающей запятой, хотя это неправильно. (Впрочем, значение может быть целым числом, если ваш возраст, вес и код города специально подобраны.) Этот недостаток устранен во врезке “Исправление IQ.C с помощью старого трюка с приведением типов”.
- ✓ Заметьте, что `getval()` определена как целочисленная функция. Внутри `getval()` целочисленное значение возвращает функция `atoi()`. Оно сохраняется в переменной `x`, и ее значение затем возвращается основной функции с помощью

инструкции `return(x)`; . Это значение — целое число, потому и функция имеет тот же тип.

- ✓ В главной функции `main()` функция `getval()` вызывается три раза. Возвращаемые ею значения сохраняются в целочисленных переменных `age` (возраст), `weight` (вес) и `height` (рост).
- ✓ Да, вероятно, вам также приходится жульничать, когда вы вводите свой вес. Чем вы толще, тем выше уровень вашего интеллекта, вычисляемый программой по заложенной в нее формуле.

Исправление IQ.C с помощью старого трюка с приведением типов

В исходном тексте IQ.C компьютер вычисляет ваш показатель интеллекта IQ по формуле:

```
iq=(age*weight)/area; // iq = (возраст * вес) / область
```

Другими словами, ваш показатель интеллекта равен вашему возрасту (`age`), умноженному на ваш вес (`weight`), а затем это произведение делится на код вашего города (`area`). Все значения этих переменных — целые числа, и, кстати сказать, именно эта формула используется в школьном округе, где учатся мои дети.

Тревога! Всякий раз, когда вы делите любые два значения, результат будет, вероятно, дробью, точнее, числом типа `float`. Да, результат будет иметь тип `float`. Это могут понять только математики. Для нормальных людей десятичные числа и дроби — это слишком запутанно.

Чтобы функция работала правильно, было объявлено, что переменная `iq` имеет тип `float` — это было сделано в начале исходного текста, как и должно быть сделано. Но есть проблема: значение, вычисленное по этой формуле, все же запоминается там, где обычно хранится целое число. (Нормально, да?) Даже если вычисленный результат имеет дробную часть, все переменные целочисленные, да и результат — целое число. (Вы помните, что компилятор ничего не предполагает?). Чтобы устранить проблему, вы должны сделать то, что называется **приведением типа**, т.е. указать компилятору, что он должен временно забыть тип переменной и вместо этого предполагать, что она имеет указанный вами тип.

Отредактируйте в исходном тексте IQ.C строку

```
iq=(age*weight)/area; // iq = (возраст * вес) / область
```

так:

```
iq=(float)(age*weight)/area;
```

Вставьте слово `float` в круглых скобках сразу после знака `"="`. Снова сохраните файл на диске. Скомпилируйте и выполните. Обратите внимание, что ваш показатель интеллекта изменится — он несколько возрастет (на дробную часть):

The computer estimates your IQ to be 27.764423.

(Компьютер считает, что ваш показатель интеллекта равен 27,764423.)

Теперь число действительно имеет тип `float`.

Возврат значения вызывающей функции с помощью ключевого слова return

Функции, возвращающие значения, нуждаются в некотором механизме, который возвращает значения назад. Информация ведь не может перевалиться через край просто так, потому что компилятор не в состоянии догадаться, что последняя фигурная скобка означает “Эй, я должен вернуть переменную, мм... х. Да. Вот именно. Возвратите х назад. Теперь я получаю ее значение”.

Нет. Чтобы должным образом возвращать значение, нужно надлежащим образом использовать ключевое слово `return` (возвратить).

Ключевое слово `return` используется, чтобы послать значение назад из функции, т.е. вернуть значение из функции. Вот формат инструкции:

```
return (возвращаемое значение);
```

Здесь *возвращаемое значение* — это то значение, которое функция должна вернуть. Какой тип должно иметь значение? Это зависит от типа функции. Значение может быть целым числом для функций типа `int`, символом для функции типа `char`, строкой (для этого понадобится специальный трюк), числом типа `float` для функции типа `float`, и так далее. Более того, можно указать имя переменной или постоянное значение.

```
return (total);
```

Значение переменной `total` (общее количество) возвращается из функции.

```
return (0);
```

Функция возвращает значение нуль.

Между прочим, *возвращаемое значение* указывать необязательно. Для функций типа `void` вы можете использовать инструкцию `return()`; , чтобы заставить программу вернуться, например, в середине выполнения какой-либо последовательности инструкций (в качестве примера см. программу `BONUS.C` ниже в этой главе).



- ✓ С технической точки зрения все функции можно заканчивать единственной инструкцией возврата `return`. Притом можно кодировать функции так, чтобы это была их последняя инструкция. Когда инструкции возврата `return` нет, компилятор автоматически вставляет возврат, когда он видит последнюю фигурную скобку функции. (Выполнение программы переваливается через край!)
- ✓ Функция, в определении которой указан тип `int`, `char` или какой либо иной, должна вернуть значение указанного типа.
- ✓ В функции типа `void` можно использовать инструкцию возврата `return`, но эта инструкция возврата не должна ничего возвращать! В функции типа `void` используйте только `return()`; или просто `return`; . В противном случае компилятор выдаст сообщение об ошибке (точнее, предупреждение).
- ✓ Если предполагается, что функция возвращает что-нибудь, но фактически возвращать нечего, используйте инструкцию возврата `return(0)`; .
- ✓ Ключевое слово `return` не обязательно должно стоять в конце функции. Иногда приходится использовать его в середине функции, как в функции `BONUS.C`, приведенной далее.

Теперь вы можете разобраться в главной функции `main()`

Во всех ваших программах и во всех программах в этой книге вы видели, что главная функция `main()` объявляется как имеющая тип `int` и всегда оканчивается инструкцией `return(0)`; . В соответствии со стандартом ANSI это основные требования языка C: главная функция `main()` должна быть определена как имеющая тип `int`, причем она должна вернуть значение.

Значение, возвращенное главной функцией `main()`, доступно операционной системе. В большинстве программ оно не используется, так что вернуть можно любое значение. Однако для некоторых программ возвращаемое значение — сигнал, который программа подает операционной системе.

Например, некоторые утилиты командной строки могут вернуть 0, если программа выполнила свою задачу. Тогда любое значение, отличное от 0, может указывать на ошибку.



- ✓ В DOS и Windows значение, возвращенное программой, точнее главной функцией `main()`, можно проверить в пакетном файле. Это значение в языке программирования пакетных файлов хранится в переменной `ERRORLEVEL` (УРОВЕНЬ ОШИБКИ).
- ✓ В операционных системах семейства Unix вы можете использовать язык сценариев (оболочку — `shell`), чтобы определить код возврата из любой программы.
- ✓ До введения стандарта ANSI главная функция `main()` обычно объявлялась как имеющая тип `void`:

```
void main()
```

Вы можете увидеть эту строку в некоторых старых книгах по программированию или в некоторых исходных текстах. Обратите внимание, что в этом нет ничего неправильного; компьютер от этого не разрушается, не взрывается и даже не выдает сообщение об ошибке. (И при этом никогда не было зарегистрировано случая, чтобы объявление главной функции `void main()` привело к какой-либо проблеме хоть на одном компьютере.) Но, несмотря на это, теперь в соответствии с новым стандартом требуется, чтобы главная функция `main()` была объявлена как имеющая тип `int`. Если вы не придерживаетесь этого правила, многие университетские второкурсники начнут показывать пальцами на вас и хихикать в разных форумах Internet. Конечно, это еще ничего не означает, но показывать пальцами на вас они будут все равно.

Премия — программа BONUS.C!

Следующая программа BONUS.C содержит функцию, которая имеет три инструкции возврата `return`. Эта программа доказывает, что вы можете вставить инструкцию возврата `return` в середине функции, и никто не будет хихикать над вами — даже университетские второкурсники:

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

```
float bonus(char x);
```

```
int main() // главная функция
{
    char name[20];
    char level;
    float b;

    printf("Enter employee name:"); // Введите имя служащего
    gets(name);
    // Введите уровень премии (0, 1 или 2):
    printf("Enter bonus level (0, 1 or 2):");
    level=getchar();
    b=bonus(level); // уровень
    b*=100;
    // ("премия для %s будет равна $ %.2f.\n", имя, b)
    printf("The bonus for %s will be $%.2f.\n",name,b);
    return(0);
}
```

/* Вычислить премию */

```
float bonus(char x)
{
    // если (x == '0') вернуть (0.33);
    if(x=='0') return(0.33); /* Основная премия */
    // если (x == '1') вернуть (1.50);
    if(x=='1') return(1.50); /* премия Второго уровня */
    /* Наивысшая премия */
    return(3.10);
}
```

Введите этот исходный текст с помощью вашего редактора. Сохраните его на диске под именем BONUS.C. Обратите внимание, что функция `bonus()` содержит три инструкции `return`, каждая из которых возвращает свое, отличное от других, значение главной функции `main()`. Кроме того, функция имеет тип `float`, который вам еще не встречался в этой книге, если вы читали главы по порядку.

Скомпилируйте и выполните.

Вот пример вывода:

```
Enter employee name:Bill
Enter bonus level (0, 1, or 2):0
The bonus for Bill will be $33.00
```

(Введите имя служащего:Bill

Введите уровень премии (0, 1, или 2):0

Премия для Bill будет равна 33,00 \$)

Выполните программу еще несколько раз с новыми именами и значениями. Попробуйте не восхищаться ее гибкостью.



✓ Бедный Билл.

✓ Вы можете испытывать искушение применить приведение типа к 100 в строке: `b *= 100;`. Но это не нужно, потому что это не переменная. Если бы число 100 хранилось в переменной типа `int`, — например, в переменной `rate` (норма), — приведение типа было бы необходимо:

```
b*=(float)rate;
```



✓ Обратите внимание, как записано значение с плавающей точкой 0.33. Значения должны всегда начинаться с цифры, а не с десятичной точки. Если записать это значение как .33, компилятор может сгенерировать сообщение об ошибке. Всегда начинайте значения с цифры — от 0 до 9.

✓ Инструкции в функции `bonus()` записаны довольно плотно. Ха! Что они делают, вы знаете...

На самом деле так оно и есть. Они действительно уплотнены, но это не означает, что вы должны писать свои программы именно так. В следующем разделе вы увидите несколько альтернативных форматов этих инструкций, которые делают то же самое. Как бы то ни было, вы должны постоянно пробовать записывать ваши программы на C короче и понятнее.

Не беспокойтесь о таких мелочах языка C, если вы спешите



C — гибкий язык, и потому в нем предлагается много способов форматирования для записи решения конкретной задачи. Рассмотрим функцию `bonus()` (премия()) из программы `BONUS.C`. Ниже представлено четыре различных способа записи этой функции, но независимо от способа функция все равно решает ту же самую задачу.

Длинный, скучный способ:

```
float bonus(char x)
{
    int v;

    if(x=='0') // если (x == '0')
    {
        v=0.33;
    }
    else if(x=='1') // иначе если (x == '1')
    {
        v=1.50;
    }
    else // иначе
    {
        v=3.10;
    }
    return(v); // вернуть (v)
}
```

Длинный, скучный способ минус лишние фигурные скобки:

```
float bonus(char x)
{
    int v;

    if(x=='0') // если (x == '0')
        v=0.33;
    else if(x=='1') // иначе если (x == '1')
        v=1.50;
    else // иначе
        v=3.10;
    return(v); // вернуть (v)
}
```

А теперь без целой переменной `v`:

```
float bonus(char x)
{
    if(x=='0') // если (x == '0')
        return(0.33); // вернуть (0.33);
    else if(x=='1') // иначе если (x == '1')
        return(1.50); // вернуть (1.50);
    else // иначе
        return(3.10); // вернуть (3.10);
}
```

Почти те же самые строки, но выкинуто лишнее.

```
float bonus(char x)
{
    if(x=='0') return(0.33);           // если (x == '0') вернуть (0.33);
    else if(x=='1') return(1.50);      // иначе если (x == '1') вернуть
(1.50);
    else return(3.10);                 // иначе вернуть (3.10);
}
```

Наконец, без else:

```
float bonus(char x)
{
    if(x=='0') return(0.33);           // если (x == '0') вернуть (0.33);
    else if(x=='1') return(1.50);      // иначе если (x == '1') вернуть
(1.50);
    return(3.10);                     // иначе вернуть (3.10);
}
```

Вы можете заменить любую предыдущую версию функции `bonus()` в вашем исходном тексте `BONUS.C` этой новой версией. Все работает просто прекрасно.

То, что пишется в начале программы

В этой главе...

- Что делает `#include`
- Создание собственных заголовочных файлов
- Как используются библиотеки
- Использование директивы `#define`
- Игнорируем макроопределения (макросы)

Вероятно, вы заметили, что в начале программ на С обычно записываются какие-то пока не очень понятные строки. Есть директивы `#include`. Могут быть директивы `#define`. Затем могут следовать прототипы функции. Возможно, определены несколько глобальных переменных. Возможно, есть некоторые комментарии. Все это находится в начале исходного текста и предшествует главной функции `main()`.

Действительно ли так должно быть?

Действительно ли это нужно?

Действительно ли это так необходимо?

Добро пожаловать в главу, в которой вы узнаете о том, что записывается в начале программ на С. Теперь, когда вы, вероятно, прочитали все другие главы и хорошо знакомы с языком С, стоит сделать некоторые выводы и описать все эти штучки. Несомненно, в этой главе рассмотрены директивы `#include`.



- ✓ Введение в директивы `#define` находится в главе 8 “Переменные в языке С”.
- ✓ Создание прототипов функций описано в главе 20 “Создание первой функции”.
- ✓ Глобальные переменные рассматриваются в главе 21 “Переменные в функциях”.
- ✓ Кроме того, в начале вашего кода на С могут встретиться и другие элементы, например, директивы, включающие объявления внешних и общедоступных переменных. Они используются в том случае, если исходный текст записан в нескольких модулях.

Пожалуйста, не пропустите меня!

Что именно делает следующая строка?

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

Это — команда для компилятора, чтобы он кое-что сделал, а точнее, включил специальный файл, имеющийся на диске, в ваш исходный текст. Этот файл называется `STDIO.H`.

На рис. 23.1 показано, что делает команда `#include <stdio.h>`. Содержимое файла `STDIO.H` читается с диска и включается (вставляется) в ваш файл исходного текста во время его компиляции.

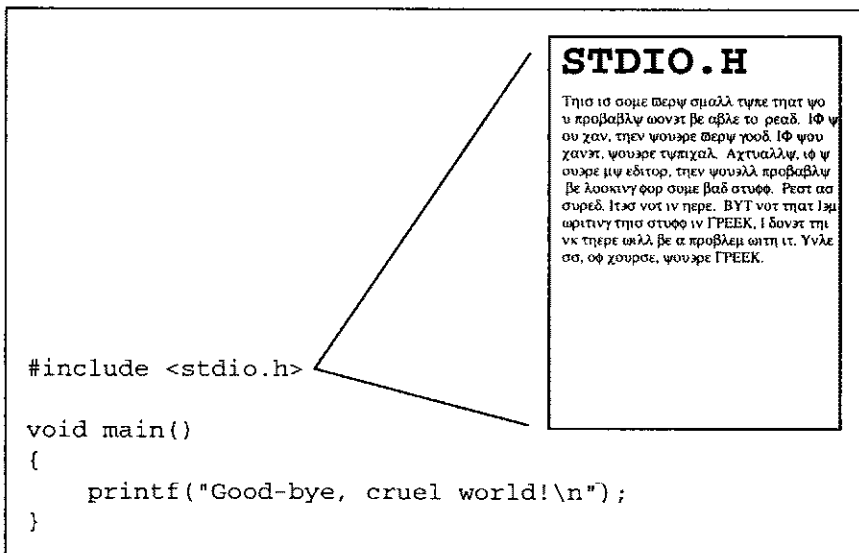


Рис. 23.1. Вот что делает директива #include в программе

На рис. 23.2 показано, что делают несколько строк, начинающихся с #include. Каждый файл читается с диска и вставляется в исходный текст, один за другим, по ходу компиляции исходного текста.

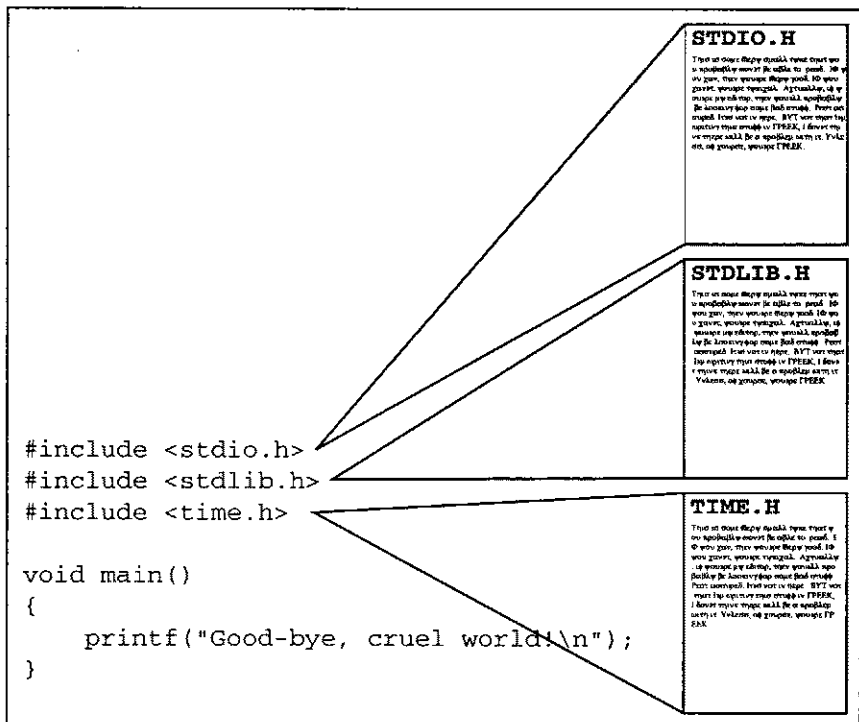


Рис. 23.2. Программа, содержащая несколько директив #include

Конструкция #include

Конструкция `#include` указывает, что компилятор должен скопировать строки из заголовочного файла в исходный текст. Эта команда нужна компилятору для того, чтобы он мог использовать многие функции языка C. Заголовочный файл содержит информацию о том, как используются функции (да, он содержит прототипы), а также другую информацию, которая помогает компилятору понять программу.

Ниже приведен формат директивы `#include`:

#include <имя файла>

После директивы `#include` следует имя файла в угловых скобках. Имя файла должно быть набрано на нижнем регистре и обычно (хотя это и не обязательно) заканчивается точкой, после которой следует строчная (маленькая) буква `h`. Подобно всем конструкциям, расположенным в начале исходного текста и начинающимся с признака `#`, эта строка не заканчивается точкой с запятой!

Иногда при необходимости имя файла представляет собой часть пути, например:

#include <sys/socket.h>

После пути `sys/` следует имя заголовочного файла `socket.h`.

Второй формат используется тогда, когда нужно включить заголовочный файл, который вы создали самостоятельно, или тогда, когда заголовочный файл не находится в каталоге INCLUDE (ВКЛЮЧАЕМЫЕ ФАЙЛЫ) компилятора:

#include "имя файла"

Формат по существу тот же самый, за исключением того, что имя файла заключено в двойные кавычки, а не в угловые скобки. В этом случае компилятор ищет заголовочный файл в том каталоге, где находится файл исходного текста.

- ✓ Заголовочные файлы помогают компилятору правильно создавать программы. Помните, что язык C содержит (приблизительно) только 32 ключевых слова (см. табл. 3.1 в главе 3 "Формальное знакомство с языком C"). Все остальные слова, вроде `printf` и `getchar`, являются именами функций. Прототипы этих функций находятся в различных заголовочных файлах, которые вы включаете в начале ваших программ. Без заголовочных файлов компилятор не сможет распознать функции и может отобразить кучу сообщений об ошибках.
- ✓ Часть имени файла `.h` означает *header* (заголовок).
- ✓ Чтобы узнать, какой заголовочный файл нужно включить для данной функции, найдите эту функцию в вашем справочном руководстве по библиотеке языка C. Заголовочный файл указан вместе с форматом функции, для которой нужен этот файл.
- ✓ Директиву включения `include` всегда записывайте на нижнем регистре. Помните, что она не заканчивается точкой с запятой и что имя заголовочного файла заключается в угловые скобки.
- ✓ Указывать заголовочный файл нужно только однажды, даже если этот файл требуется для двух различных функций.
- ✓ Вы, вероятно, видели, что сообщают некоторые компиляторы, если включить заголовочный файл. Например, компилятор может сообщить "о 435 откомпилированных строках", даже если ваш исходный текст содержит только 20 или 30 строк. Дополнительные строки считываются из файла (или файлов), включаемого директивой `#include`.





- ✓ Не обязательно указывать полный путь к заголовочному файлу. Дело в том, что компилятор знает, где находятся заголовочные файлы. Заголовочные файлы расположены в специальном подкаталоге, создаваемом на вашем жестком диске во время установки компилятора языка C. Подкаталог называется INCLUDE, и именно он содержит все файлы *.H, поставляемые с вашим компилятором. Все такие файлы являются текстовыми, и вы можете просмотреть их, используя средства просмотра файлов или ваш редактор. (Пожалуйста, не изменяйте эти файлы!)
- ✓ В Windows папка INCLUDE находится ниже папки, в которой был установлен ваш компилятор.
- ✓ В операционных системах семейства Unix папка INCLUDE находится в каталоге /usr: /usr/include.



Компилятор и препроцессор: длинный и нудный рассказ, который вы можете пропустить

Я когда-то рассказывал вам, как текстовый файл становится программой. Помните раздел, посвященный компиляторам? Вероятно, нет. Так или иначе, я там солгал. Я тогда опустил шаг, который был лишним на том этапе знакомства с языком C. Прежде чем компилятор начнет компилировать, некая штука, называемая препроцессором, быстренько просматривает исходный текст и выполняет кое-какие замены в нем.

Препроцессор ничего не компилирует. Вместо этого он ищет в исходном тексте строки, начинающиеся с признака фунта (#). Эти строки — скрытые команды препроцессора, они указывают ему, что нужно кое-что сделать.

Компилятор распознает несколько директив препроцессора. Наиболее часто встречаются директивы `#include` и `#define`. Другие представляют собой макроопределения (макросы), которые указывают компилятору, нужно ли компилировать некоторые части исходного текста — что-нибудь вроде "Если выполнено такое и такое условия, то игнорировать следующие несколько строк". Впрочем, все, что я рассказывал вам о препроцессоре, вы можете полностью игнорировать, если не собираетесь стать суперзнатоком языка C.

Что такое STDIO.H?

Обычно директива `#include <stdio.h>` заставляет компилятор искать на диске файл по имени `STDIO.H` и копировать каждую строку из этого файла в ваш исходный текст. Нет, вы не увидите скопированные строки; ведь это происходит в процессе преобразования исходного текста в объектный код. (См. врезку "Компилятор и препроцессор: длинный и нудный рассказ, который вы можете пропустить".) `STDIO.H` — стандартный заголовочный файл ввода-вывода, из которого считывается все, связанное с `stdio`. (Это совсем не "студия-h", хотя многие программисты читают это слово именно так.) В `STDIO.H` определены все прототипы для стандартных команд ввода-вывода, таких как `printf()` и `puts()`. Кроме того, он содержит определения для других часто встречающихся вещей в C, опции компилятора и всякое другое барахло.

Вы можете просмотреть этот файл и его содержимое, поскольку вы можете просмотреть любой заголовочный файл. Просто откройте папку INCLUDE на жестком диске вашего компьютера и используйте команду просмотра файла.

В Windows используйте следующую команду:

```
type stdio.h | more
```

В операционных системах семейства Unix выполните вот эту команду:

```
less stdio.h
```

Найдите в файле ваших старых друзей, вроде `printf()`, чтобы увидеть, как они определены и какой у них прототип в заголовочном файле.

Пишем свой собственный файл с расширением .h

Нет абсолютно никакой потребности писать свой собственный заголовочный файл. Ни теперь, ни когда-либо в будущем. Конечно, если вы хотите стать гуру по языку C, вы можете однажды написать монстр-мультимодуль и создать для него собственный заголовочный файл, чтобы с гордостью показать его вашим товарищам-специалистам по C и, вполне возможно, они внимательно рассмотрят его и с сожалением скажут: "Мне жаль, что я не могу делать такое". Чтобы сделать это, и чтобы эта глава не была слишком короткой, создайте в вашем редакторе заголовочный файл `HEAD.H`. Напечатайте строки точно так, как написано, строго соблюдая нижний и верхний регистры:

```
/* Это - мой крошечный заголовочный файл */
// СЧАСТЛИВЧИК

#define HAPPY 0x01
// ВЫДАТЬ

#define BELCH printf
// ОТКРЫТЬ

#define SPIT (
// ЗАКРЫТЬ

#define SPOT )
```

Сохраните файл на диске под именем `HEAD.H`. Вы должны напечатать `.H`, чтобы ваш редактор не сохранил этот файл на диске с расширением `C`, `TXT` (ТЕКСТ) или `DOC`.

Компилировать или "выполнять" заголовочный файл не нужно. Вместо этого вы вставляете его с помощью `#include` в ваш исходный текст. Затем следует несколько своеобразная программа. Она может вовсе не походить на обычную программу на C. То, что вы увидите ниже, может вызвать отвращение, возможно, что вы захотите выбросить эту книгу, даже не дочитав до конца. Пожалуйста, я прошу, отложите свое решение и не спешите в магазин покупать книгу по Visual Basic:

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

int main() // главная функция
SPIT // ОТКРЫТЬ
// ВЫДАТЬ ("Этот парень счастлив: %c\n ", СЧАСТЛИВЧИК);
BELCH("This guy is happy: %c\n",HAPPY);
return(0);
SPOT // ЗАКРЫТЬ
```

Начните с чистого документа в вашем редакторе. Напечатайте предыдущий исходный текст в точности так, как он приведен выше. Вторая директива `#include` вводит ваш заголовочный файл; обратите внимание, что имя `head.h` заключено в двойные кавычки. Это сделано для того, чтобы компилятор мог найти ваш заголовочный файл, а не искал его среди других традиционных заголовочных файлов. Кроме того, пусть вас не смущают слова `SPIT` (ОТКРЫТЬ), `BELCH` (ВЫДАТЬ), `SPOT` (ЗАКРЫТЬ). Я объясню их позже.

Сохраните файл на диске и назовите его `HTEST.C`. Скомпилируйте и выполните. Не падайте в обморок, если не увидите никаких ошибок. Хотите — верьте, хотите — нет, но все должно прекрасно работать. Вот как будет выглядеть вывод:

```
This guy is happy:0
```

```
(Этот парень счастлив:0)
```

Мистер СЧАСТЛИВЧИК счастлив. Вы также можете быть счастливы, так что продолжим разбор программы.

Вторая директива `#include` вставляет в ваш исходный текст `HEAD.H` файл, который вы создали ранее. Все команды из этого файла волшебным образом включаются в ваш исходный текст, когда компилируется `HTEST1.C` (точно так же, как и команды из стандартного заголовочного файла ввода-вывода `STDIO.H`).

Вспомните, что в заголовочном файле `HEAD.H` есть несколько тех директив `#define`, которые велит компилятору заменить некоторыми символами или словами языка `C` эвфемизмы счастливчика. (Просмотрите ваш файл `HEAD.H` еще раз, если нужно.) Например, в определении слова `SPIT` было указано, что оно равно левой фигурной скобке. Поэтому в начале программы используется `SPIT`, а не фигурная скобка в строке:

```
SPIT
```

```
// ОТКРЫТЬ
```

В определении слова `BELCH` было указано, что оно равно слову `printf`, так что `BELCH` служит заменой (как бы является заместителем) этой функции в строке `BELCH("This guy is happy: %c\n", HAPPY);`, а в последней строке слово `SPOT` заменяет заключительную фигурную скобку `}`.

Почти все, что допускается в программе на языке `C`, допускается и в ваших собственных заголовочных файлах. Наиболее популярны там инструкции `#define`. В заголовочных файлах допускаются также комментарии, определения переменных, структуры и даже исходные тексты (хотя встречаются они там редко). Помните, что поскольку в конечном счете заголовочные файлы копируются в ваш исходный текст, все, что обычно встречается в нем, может быть записано также в заголовочном файле.



- ✓ Как правило, любой заголовочный файл, написанный программистом, содержит много директив `#define`. Можно записать директивы `#define` для необычных комбинаций клавиш, например:

```
#define F1 0x3B00
```
- ✓ Эта строка позволяет мне использовать символ, соответствующий клавише `<F1>`, а не помнить (или постоянно искать) значение, которое в компьютере замещает клавишу `<F1>`.
- ✓ Не можете запомнить код счастливого лица? Запишите `#define` в вашем собственном заголовочном файле и используйте это определение вместо кода этого символа — именно так сделано в предыдущей программе.
- ✓ Не стоит переопределять слова и функции языка `C` (если вы, конечно, не хотите удивить свою тетушку). Однако первая директива `#define` в `HEAD.H` устанавливает слово `HAPPY` (СЧАСТЛИВЧИК) равным значению символа счастливого лица в ПК, т.е. `0x01`.

Последнее предупреждение относительно заголовочных файлов



Типичный заголовочный файл, используемый в `C`, содержит много информации. Однако обычно он не содержит код программы. Под этим я подразумеваю, что, хотя в файле `STDIO.H` может быть определена функция `printf()` и этот файл может содержать ее прототип, объяснение системы сообщений и даже разнообразные указания по ее использованию, код программы, которая выполняет задачу, возложенную на `printf()`, не находится в заголовочном файле.

Иной читатель пишет мне: “Дэн! Мне нужен заголовочный файл `DOS.H`, чтобы скомпилировать старую программу для `DOS`. Вы можете найти этот файл и выслать его мне?” Вот мой ответ: даже если бы я мог найти этот заголовочный файл, один он не поможет, вам также нужен библиотечный файл, чтобы выполнить задание.

Библиотечные файлы подобны объектному коду. Они содержат команды для микропроцессора, которые указывают, как выполнить ту или иную задачу. Команды для функции `printf()`, выполняющие возложенную на нее задачу, содержатся в стандартном библиотечном файле языка C, а не в заголовочном файле.

На рис. 23.3 показано, как все это происходит. Сначала вы берете ваш исходный текст, например `HOLLOW.C`. Затем во время компиляции директива `#include` вставляет в исходный текст файл `STDIO.H`. Результатом компиляции является объектный файл по имени `HOLLOW.O` (или `HOLLOW.OBJ`).

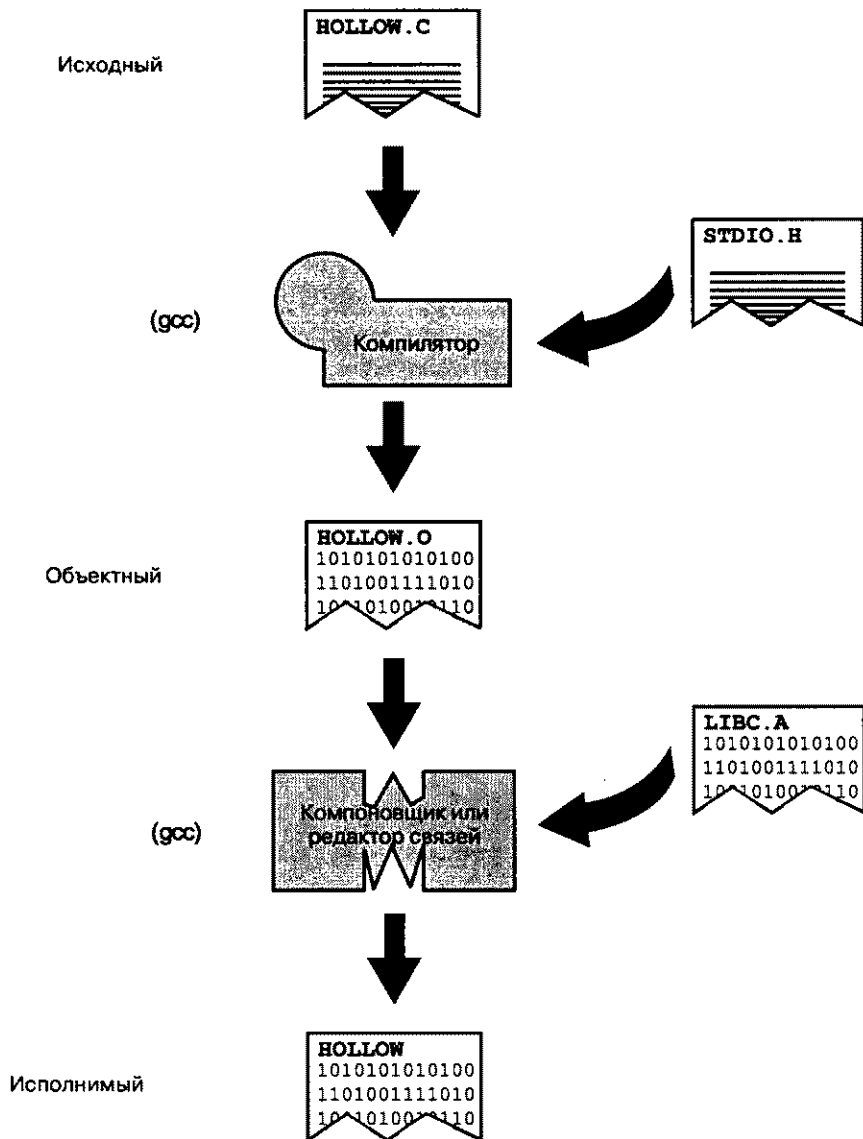


Рис. 23.3. Использование библиотечных файлов

Чтобы превратить объектный файл в программный (исполнимый) файл, добавляется библиотека. Библиотека содержит команды для функций языка С. Они аккуратно *компонуются* с вашим исходным текстом (точнее, с объектным кодом, полученным из него), чтобы получилась исполнимая программа HOLLOW (ПУСТОТА) (или HOLLOW.EXE).

Код содержит именно библиотека, а не заголовочный файл! Если требуется скомпилировать старую программу для DOS, нужна старая библиотека для DOS, чтобы выполнить компоновку с объектным кодом, полученным из вашего исходного текста. Без этой библиотеки заголовочный файл не поможет.

- ✓ Библиотечные файлы живут в папке LIB, которая обычно находится там же, где и папка INCLUDE (ВКЛЮЧАЕМЫЕ ФАЙЛЫ). Например, в Unix папка /usr/lib находится в оглавлении /usr — там же, где и папка /usr/include.
- ✓ Подробно использование библиотечных файлов описано в книге *C All-in-One Desk Reference For Dummies*, выпущенной издательством “Wiley”.

Что такое #define

Директиву #define компилятор обрабатывает до того, как приступит к вашему исходному тексту. Используя #define, вы можете установить некоторые удобные, умные и незабываемые слова-эквиваленты для больших чисел, постоянных значений, да и практически для всего, что может встретиться в вашем исходном тексте на языке С.

Полное описание “конструкции” #define приведено в главе 8 “Переменные в языке С”. Я здесь не собираюсь повторять приведенные там сведения, однако отмечу, что #define — наиболее часто встречающаяся директива в заголовочных файлах. Отредактируйте файл HEAD.H и добавьте эти строки:

```
// ИСТИНА
#define TRUE 1
// ЛОЖЬ (! ИСТИНА)
#define FALSE (!TRUE)
```

Сохраните файл HEAD.H с этими модификациями на диске.

Вы создали слово-сокращение TRUE (ИСТИНА), которое можно использовать как “истинное” условие.

Слово-сокращение FALSE (ЛОЖЬ) определено как “НЕ ИСТИНА”. Восклицательный знак в языке С означает “НЕ”. Значение FALSE, очевидно, можно определить как !TRUE. (Это значение на самом деле равно 0.)

Чтобы немедленно использовать эти два новых слова-сокращения, измените исходный текст HTEST.C:

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

int main() // главная функция
SPIT // ОТКРЫТЬ
    if (TRUE) // если (ИСТИНА)
        // ВЫДАТЬ ("Должно быть истинно!\n")
        BELCH("Must be true!\n");
    if (FALSE) // если (ЛОЖЬ)
        // ВЫДАТЬ ("Никогда не увидите этого сообщения.\n");
        BELCH("Never see this message.\n");
    return(0);
SPOT // ЗАКРЫТЬ
```

Тщательно напечатайте этот исходный текст с помощью вашего редактора. Сохраните его на диске. Скомпилируйте и выполните.

Вывод будет выглядеть так:

Must be true!

(Должно быть истинно!)



- ✓ Слова-сокращения TRUE и FALSE очень удобно использовать для создания цикла с условием продолжения. Обратитесь к главам 17 “Познакомьтесь с циклом while (циклом с условием продолжения)” и 18 “Циклы с условием продолжения. Организация задержки”.

- ✓ В языке C значение 1 представляет собой TRUE, а 0 — FALSE.

- ✓ Некоторые программисты в языке C предпочитают определять TRUE и FALSE так:

```
#define FALSE 0
#define TRUE (!FALSE)
```

В этой книге TRUE определяется как 1, а FALSE как “не истина”, что оказывается равным 0. Если в программе используются предыдущие директивы #define, все работает так, как надо. Можно не беспокоиться.

- ✓ Некоторые компиляторы могут сгенерировать предупреждающее сообщение, если условие в круглых скобках условного оператора отсутствует. Это просто хитрость компилятора; он распознает условия-константы в условии ключевого слова if и сообщает вам, тождественно истинно (равно TRUE) это условие или оно всегда ложно (равно FALSE).
- ✓ Некоторые компиляторы могут также указать, что второй условный оператор никогда не выполняется, потому что условие в if всегда ложно (равно FALSE). Но это всего лишь предупреждения. Выполните программу так или иначе.

Макроопределения (макросы) — эти темы мы даже не обсуждаем



В языке C с # начинаются также макрокоманды. Подобно #include и #define, макроопределение (макрос) — команда компилятору, которая указывает ему, как нужно обработать условия и следует ли компилировать определенные части исходного текста или их нужно пропустить. Необычно, но удобно.

Совсем нет необходимости подробно описывать, что такое макроопределение (макрос), и, более того, тратить впустую время на примеры. Тем не менее, я приведу список слов, начинающихся со знака фунта, поскольку они могут встретиться в макросах в исходных текстах некоторых программ на языке C. Вот этот список:

- ✓ **#if**
- ✓ **#else**

- ✓ #endif
- ✓ #ifdef
- ✓ #ifndef

Эти слова также являются командами для компилятора с языка С, предназначенными прежде всего для того, чтобы указать, должна ли быть скомпилирована та или иная группа инструкций, т.е. для того, чтобы указать, нужно ли включать ее в окончательную программу.

Например:

```
// ГРАФИКА
#ifdef GRAPHICS
// здесь сделать графические чудеса
#else
// здесь вывести скучный текст
#endif
```

Предположим, что слово GRAPHICS определено ранее в программе или в заголовочном файле — например, так #define GRAPHICS 1. Если так, компилятор компилирует инструкции только между #if GRAPHICS и строкой #else. В противном случае компилируются инструкции между #else и #endif. Результат? Из одного исходного текста получаются две различных программы.

Сумасшедший! О чем я беспокоюсь?!

Я использовал эту уловку только однажды. Я написал программу, которая имела две версии — ту, которая выполняется на старых ПК с цветными мониторами, и вторую, которая выполняется на ПК с монохромными (одноцветными, т.е. черно-белыми) мониторами. Для ПК с цветными мониторами я написал инструкции, которые отображают текст в цвете, но эти инструкции не выполнялись на старых монохромных (одноцветных, т.е. черно-белых) мониторах. Поэтому я использовал директивы #if так, чтобы записать только один файл исходного текста, из которого можно создать обе версии программы.

- ✓ Действительно, нет никакой потребности запоминать эти директивы. Я расскажу о них только потому, что обещал это сделать в заголовке раздела, в противном случае некоторые профессора, преподающие академический курс языка С, отчитали бы меня за игнорирование этой темы.
- ✓ После #if следует слово-сокращение, определенное в программе в другом месте с помощью конструкции #define. Если значение этого слова не равно 0, условие #if конструкции истинно, и компилируются инструкции между #if и #endif. О, есть еще слово #else, которое играет ту же самую роль, что и else (иначе) в языке С.
- ✓ Вы можете встретить конструкции #ifdef или #ifndef в различных заголовочных файлах. Я понятия не имею, что они там делают, но выглядят они весьма внушительно.
- ✓ Есть еще одна конструкция, которая начинается с # — #line. Она используется редко. Данная конструкция заставляет компилятор вставить уникальный номер строки в сообщение об ошибке. (Не волнуйтесь — я также не использую эту конструкцию.)



Глава о функции printf()

В этой главе...

- Использование `printf()` для отображения текста
- Отображение запрещенных символов с помощью escape-последовательностей
- Отображение значений переменных
- Символы преобразования

В

ероятно, в программах на C чаще всего встречается функция `printf()`. Хотя вы, возможно, использовали ее, но несмотря на мои различные подсказки в отношении ее мощи я думаю, что по-настоящему вы с ней не знакомы. Теперь пришло время изучить `printf()` основательно.

Краткий обзор printf()

Вы должны знать о некоторых особенностях `printf()`.

- ✓ Функция `printf()` отображает текст, который заключен в двойные кавычки.
- ✓ Чтобы отобразить некоторые специальные символы, в `printf()` используются escape-последовательности — последовательности символов, начинающиеся с наклонной черты влево.
- ✓ Функция `printf()` может отобразить значения переменных, для этого применяются символы преобразования, начинающиеся с `%`.
- ✓ Для форматирования выводимой информации могут также использоваться символы преобразования, начинающиеся с `%`.

Старая подпрограмма printf() — отображение текста

Функция `printf()` предназначена для отображения текста на экране. Вот основной формат:

```
printf("текст");
```

Здесь *текст* — это тот текст, который вы хотите увидеть на экране. Он заключен в двойные кавычки. Двойные кавычки заключены в круглые скобки, а вся инструкция должна заканчиваться точкой с запятой.

- ✓ Специальные символы — вроде двойной кавычки, табуляции, возврата на один символ и новой строки — можно включить в текст, отображаемый `printf()`. Такие символы в `printf()` представляются escape-последовательностями (об этом читайте в следующем разделе).

- ✓ Функция `printf()` может отобразить несколько строк текста с помощью escape-последовательности `\n` (newline — новая строка).
- ✓ Чтобы закодировать двойную кавычку в текстовой строке, используйте escape-последовательность `\"`.

Escape-последовательности в функции `printf()`

В табл. 24.1 указаны многие из escape-последовательностей, используемых в функции `printf()`. С большинством из них вы уже знакомы, потому что использовали их. Другие очень специфичные, и вы навряд ли будете использовать их в ваших программах.

Таблица 24.1. Escape-последовательности в функции `printf()`

Последовательность	Сокращение или эквивалент
<code>\a</code>	Гудок динамика
<code>\b</code>	Возврат на один символ — Backspace (возвращает курсор обратно без стирания)
<code>\f</code>	Перевод страницы — Form feed (прогоняет страницу на принтере; может очистить экран на некоторых компьютерах)
<code>\n</code>	Новая строка (newline), действует подобно нажатию клавиши <Enter>
<code>\r</code>	Перевод каретки (перемещает курсор в начало строки)
<code>\t</code>	Позиция табуляции (Tab)
<code>\v</code>	Вертикальная позиция табуляции (перемещает курсор вниз на одну строку)
<code>\\</code>	Символ наклонной черты влево
<code>\'</code>	Апостроф
<code>\"</code>	Символ двойной кавычки
<code>\?</code>	Вопросительный знак
<code>\0</code>	Нулевой (null, пустой) байт (это 0, а не символ (буква) O)
<code>\Onn</code>	Символьное значение в восьмеричном коде (основание системы счисления 8)
<code>\xnnn</code>	Символьное значение в шестнадцатеричном коде (основание системы счисления 16)

Заключительные два символа в табл. 24.1 самые гибкие. Они позволяют вставлять любой символ в любой текст, если вы знаете восьмеричный или шестнадцатеричный код символа.

Предположим, что вы должны отобразить символ ESC (Escape) в тексте. Заглянув в приложение Б “Таблица ASCII”, вы найдете шестнадцатеричный код ESC — 1b. Поэтому нужно написать escape-последовательность — escape-последовательность для ESC!

```
\x1b
```

Вот как это выглядело бы в `printf()`:

```
// На некоторых мониторах это очищает экран
printf("On some consoles, this clears the screen \x1b[2J");
```

Вы можете сделать закладку на странице с табл. 24.1. Страница содержит escape-последовательности, которые мало кто помнит, так что к табл. 24.1 обращаться приходится весьма часто.

Испытание escape-последовательности в роскошной программе с функцией printf()

Чтобы увидеть, что означают некоторые символы, создайте приведенную чуть ниже программу PRINTFUN.C. Изменяя инструкцию printf() в ядре программы, вы сможете продемонстрировать, как различные escape-последовательности влияют на текст:

```
/* демонстрация escape-последовательности в программе с printf() */
#include <stdio.h>
int main()                                // главная функция
{
    // ("Вот последовательность \\a: \a")
    printf("Here is the \\a sequence: \a");
    getchar();
    return(0);
}
```

Введите эту программу с помощью вашего текстового редактора. Сохраните ее на диске под именем PRINTFUN.C.

Скомпилируйте и выполните PRINTFUN.C. Цель программы состоит в том, чтобы вы смогли увидеть, как последовательность \a "появляется" в тексте, т.е. отображается. Вот пример вывода программы:

Here is the \a sequence: **гудок!**

Гудит, конечно, динамик. Как ужасно! Молите Господа избавить вас от проверки этой программы ранним утром, так как в противном случае вы можете разбудить соседей.

- ✓ Во время выполнения программы getchar() ждет нажатия клавиши на клавиатуре. Эта пауза позволяет изучить вывод перед выходом из программы.
- ✓ Текстовая строка в printf() содержит две escape-последовательности. Первая — \\, двойная наклонная черта влево, которая отображает символ наклонной черты влево. В этом примере \\a отображается как \a — это и есть проверяемая escape-последовательность. Вторая escape-последовательность находится в конце строки, это \a.
- ✓ Обратите внимание, что строка в printf не заканчивается escape-последовательностью \n. Символ новой строки (newline) мог бы испортить эффект от некоторых escape-последовательностей (\r, \t и \b).

Испытание PRINTFUN

Чтобы по-настоящему испытать программу PRINTFUN, ее нужно отредактировать заново: заменить \\a и \a другой escape-последовательностью. Таким способом вы можете проверить все последовательности и почувствовать, что делает каждая из них.

Начните с того, что замените строку

```
printf("Here is the \\a sequence: \a");
```

следующей:

```
printf("Here is the \\b sequence:\b\b\b\b");
```

Эта строка проверяет escape-последовательность \b, возврат на один символ. Сохраните измененную программу, скомпилируйте и выполните ее.

Вы увидите, что курсор будет находиться под *и* в *sequence* (последовательность), когда программа выполнится. Дело в том, что \b передвигает курсор назад, но ничего не стирает.

Здесь четыре вхождения \b, которые передвигают курсор на четыре позиции назад от конца строки. (Если курсор находится в другом месте, вы, вероятно, в программе напечатали лишний пробел, или же указали больше или меньше escape-последовательностей \b.) С сим-

волом \n вы знакомы, но что делает \r? Чем перевод каретки отличается от новой строки? Чтобы узнать это, отредактируйте ту же строку в PRINTFUN.C так:

```
printf("Here is the \\r sequence: \r");
```

Сохраните измененную программу на диске, скомпилируйте и выполните.

В выводе вы увидите, что курсор вспыхнул под H в начале строки. Перевод каретки напоминает перевод каретки на пишущей машинке: этот символ перемещает курсор в начало строки. Только нажав перевод строки на пишущей машинке, можно продвинуть страницу.

Символ \t вызывает переход на позицию табуляции подобно нажиму клавиши табуляции. Курсор перемещается на позицию, которая расположена на расстоянии, равном некоторому предопределенному числу символов, причем отсчет символов ведется слева, т.е. от начала строки. Этот символ идеально подходит для выравнивания таблиц, в которых текст должен быть выстроен в колонки. Отредактируйте в программе ту же строку так:

```
printf("Able\tBaker\tCharlie\n");
```

Затем вставьте следующие строки сразу после предыдущей инструкции printf():

```
printf("1\t2\t3\n");  
printf("Alpha\tBeta\tGamma\n");
```

Никакие пробелы в этой текстовой строке для printf не нужны. Слова Able, Baker и Charlie отделены escape-последовательностями \t (позиция табуляции). Строка кончается символом \n (newline, новая строка). То же самое справедливо и для двух новых строк: символы \t отделяют числа (номера) и слова.

Перепроверьте ваш исходный текст! Убедитесь, что \t дважды встречается в каждой инструкции printf() и что \n заканчивает каждую строку текста, заключенную в двойные кавычки. Остерегайтесь лишних наклонных черт влево, которые имеют тенденцию незаметно проскальзывать в печатаемый текст при вводе таких сложных строк. Когда все будет выглядеть правильно, сохраните файл исходного текста PRINTFUN.C на диске. Скомпилируйте его. Выполните его. Вот как будет выглядеть вывод:

```
Able    Baker    Charlie  
1       2       3  
Alpha   Beta    Gamma
```

Хотя \t в инструкциях printf выглядят неряшливо, вывод хорошо организован. Вот это настоящая таблица!

- ✓ В языке C “табуляторы” предварительно установлены на каждый восьмой столбец. Использование \t вставляет необходимое количество пробелов в вывод, выстраивая в столбец следующую порцию текста, когда встречается следующий табулятор. Я подчеркиваю это, потому что некоторые люди предполагают, что позиция табуляции всегда перемещает курсор на восемь (или другое количество) символов. Но это не так.
- ✓ Символы \f и \v отображают на дисплее специальные символы в приглашении к вводу команды Windows. \f отображает символ якоря, а не перевод страницы. А \v отображает символ >, а не вертикальную позицию табуляции.
- ✓ Если вы знаете шестнадцатеричное значение кода символа, вы можете отобразить его с помощью escape-последовательности \x. Достаточно вставить шестнадцатеричный код!

Полный формат printf()

Функция printf() может также отобразить содержимое переменных. В этой книге повсюду встречались многочисленные примеры отображения содержимого целочисленных переменных с помощью метки-заполнителя %d, символьных переменных — с помощью метки-заполнителя %c, и так далее. В этом случае используется следующий формат для printf():

```
printf("format_string",[var,...]);
```

Текст *format_string*, заключенный в двойные кавычки, отображается на экране, но при отображении значений переменных он рассматривается как строка формата. (Конечно, все равно это тот же самый текст в двойных кавычках.) Ну, а после строки формата следует одна или несколько переменных; первая из этих переменных обозначена *var* (переменная величина).

Значения этих переменных, условно обозначенных *var*, вставляются в *format_string* вместо специальных меток-заполнителей, начинающихся со знака процента. Эти метки-заполнители, начинающиеся со знака процента, называются символами преобразования. Например:

```
// ("Да, я думаю, что %s также безобразник.\n", jerk)
printf("Yeah, I think %s is a jerk, too.\n",jerk);
```

Строка формата — это тот текст, который `printf()` отображает на экране: *Yeah, I think ____ is a jerk, too.* (Да, я думаю, что ____ также безобразник.). *%s* — символ преобразования — нечто вроде пустого места, которое в этом бланке нужно заполнить строкой текста. (Я называю эти символы метками-заполнителями, но знатоки С утверждают, что они называются символами преобразования.) После строки формата следует запятая и затем *jerk*. *jerk* — строковая переменная, содержимое которой заменяет *%s* в строке, выводимой функцией `printf()`.

- ✓ В строке формата функции `printf()` можно указать любое количество символов преобразования. Каждый символ преобразования, однако, должен иметь соответствующую ему переменную. Например, для трех символов *%s* требуется три строковых переменных.
- ✓ Да, это подобно заполнению бланка; символы преобразования, начинающиеся с %, — пробелы в заполняемом бланке.
- ✓ Используя надлежащие символы преобразования, можно отобразить и строки текста, и числа. Это описано в следующем разделе.
- ✓ В главе 4 “Что такое ввод-вывод?” на рис. 4.2 показано, как в инструкции `printf()` символы преобразования заменяются значениями переменных.

Символы преобразования, используемые в функции `printf()`

В табл. 24.2 приведены все символы преобразования, используемые в функции `printf()`, — даже те, которые вам никогда не понадобятся.

Таблица 24.2. Символы преобразования, используемые в функции `printf()`

Символ преобразования	Отображает параметр (содержимое переменной) как
<i>%c</i>	Отдельный символ (<i>char</i>)
<i>%d</i>	Десятичное (decimal) целое число со знаком (<i>int</i>)
<i>%e</i>	Значение с плавающей точкой со знаком в экспоненциальной нотации
<i>%f</i>	Значение с плавающей точкой со знаком (<i>float</i>)
<i>%g</i>	Значение со знаком в формате <i>%e</i> или <i>%f</i> , в зависимости от того, какое представление короче
<i>%i</i>	Десятичное целое число со знаком (<i>int</i>)
<i>%o</i>	Восьмеричное (octal) целое число без знака (основание 8) (<i>int</i>)

Символ преобразования	Отображает параметр (содержимое переменной) как
<code>%s</code>	Строка текста (string)
<code>%u</code>	Десятичное целое число без знака (unsigned int)
<code>%x</code>	Шестнадцатеричное (hexadecimal) целое число без знака (основание 16) (int)

Кроме символов преобразования, перечисленных в табл. 24.2, есть еще три символа: `%p`, `%n` и `%%`. Символы преобразования `%p` и `%n` не рассматриваются в этой книге. `%%` просто отображает `%` на экране.

- ✓ Символы преобразования, как и escape-последовательности, используются весьма часто, но все равно их никто никогда не помнит. Я советую сделать закладку на странице с этими символами.
- ✓ Символы преобразования `%x`, `%e` и `%g` имеют эквиваленты на верхнем регистре: `%X`, `%E` и `%G`. Используя символы преобразования с заглавными буквами, а не с буквами на нижнем регистре, вы гарантируете, что в выводе все символы, отображаемые на месте символов преобразования, также будут на верхнем регистре. Предположим, например, что `%x` отображает шестнадцатеричное значение как `1ba2`, тогда `%X` отобразит его как `1BA2`. Иными словами, регистр отображаемых символов тот же самый, что и регистр символа преобразования.
- ✓ Символ `%r` используется для печати указателя.

Дополнительные возможности форматирования

Руководство по компилятору с языка C должно содержать список дополнительных средств форматирования информации с помощью `printf()`, т.е. в нем должны быть перечислены дополнительные символы, которые могут использоваться вместе с символами преобразования, чтобы поддержать средства дополнительного форматирования вывода `printf()`. Эта информация слишком сложна и детализирована, чтобы приводить ее в этой книге. Кроме того, эти дополнительные средства специфичны для каждого компилятора. Я советую вам поискать `printf` в вашей интерактивной справочной системе и обратить внимание на следующие разделы в описании средств форматирования:

- ✓ флажки (flags);
- ✓ спецификаторы ширины (width specifiers);
- ✓ спецификаторы точности (precision specifiers);
- ✓ модификаторы входного размера (input-size modifiers).

На текущем этапе изучения языка программирования C заучивать эту информацию не нужно. Однако эти дополнительные средства весьма удобно использовать тогда, когда нужно, чтобы числа или другая информация в выводе `printf()` выглядела чуть более аккуратно, чем в этой главе.

Математическое безумие!

В этой главе...

- Использование математических функций
- Функции `pow()` и `sqrt()`
- Связывание с математической библиотекой в Unix
- Выполнение увеличения (инкремента) и уменьшения (декремента): до и после

Большинство людей думает, что программирование — это сплошная математика. Это неправильно. Что и говорить, в нем действительно используется математика, но самую уютную часть — вычисление ответа — делает компьютер. Так что же является самым важным? Нужно ли истерически кричать и в панике бежать от этой невинной, доброй небольшой главы, в которой просто обсуждаются различные (в основном математические) аспекты языка C?

Больше о Математике

К настоящему времени вам должны быть знакомы основные четыре компьютерных математических символа:

- ✓ символ сложения: +
- ✓ символ вычитания: -
- ✓ символ умножения: *
- ✓ символ деления: /

Эти обозначения операций обычны повсюду в компьютерной математике. Эти же символы, например, используются в электронных таблицах для обозначения математических действий. Все эти символы образуют группы на цифровой клавиатуре. Самый загадочный — звездочка (*), обозначающий умножение.

Кроме символов, из математических сведений, о которых нужно помнить при изучении языка C, — только то, что вычисления всегда указаны *справа* от знака “=”. Например:

```
meals=breakfast+lunch+dinner;
```

Если `meals`, `breakfast`, `lunch` и `dinner` — числовые переменные, то это равенство вполне законно. Однако следующая инструкция ошибочна:

```
breakfast+lunch+dinner=meals;
```

Математические действия указываются справа от знака “=”. Всегда.

- ✓ Если неправильно записать математическое равенство, получите неприятные сообщения об ошибках. Скорее всего, в них будет говориться о неправильном использовании `Lvalue` (именующее выражение или адрес переменной).



- ✓ Чтобы вспомнить загадочные математические символы, применяемые в языке C, посмотрите на цифровую клавиатуру — все они там есть.
- ✓ Математические действия в C выполняются слева направо. Некоторые операции, однако, имеют приоритет над другими. Если вы забыли, вспомните Мою Дорогую Тетушку Салли из главы 11 “Больше математики и Священный порядок (старшинство операций)”.
- ✓ Есть еще один математический символ — %, он означает остаток (по модулю). Эта ужасная тема рассматривается в главе 26 “Старая функция генерации случайных чисел”, там же приведены соответствующие примеры.
- ✓ Почти каждый необычный символ на вашей клавиатуре обозначает какой-то оператор языка C. К счастью, запоминать их все совсем не обязательно.

Возводя математические проблемы в более высокую степень

Однако есть две математические операции, для которых, кажется, не предусмотрены символы — возведение в степень и извлечение квадратного корня. Для обозначения этих операций специальные символы не предусмотрены. Конечно, я предполагаю, что вы когда-то слышали об этих операциях, но ведь неизвестно, в каких задачах они могут появиться совершенно неожиданно.

Возведение в степень встречается в задачах, в которых содержатся слова *в квадрате*, *в кубе* и *в степени*. Например:

“Четыре в квадрате” означает 4×4 , или 4^2 . Последнее читается также “четыре во второй степени”.

“Четыре в кубе” означает $4 \times 4 \times 4$, или 4^3 . Это читается также “четыре в третьей степени”.

Если речь идет не о кубе (третьей степени), или квадрате, то указывается степень; так, $4 \times 4 \times 4 \times 4 \times 4$ — это 4^5 , или “четыре в пятой степени.” (Даже не пытайтесь вычислить ответ; компьютер сделает это за Вас!)

Увы, в C нет никакого удобного способа записать 4^5 . Нет такого символа, который бы начал возведение в степень. Например, вы не можете записать

```
answer=4♠5;
```

Эта строка не годится, ведь символа ♠ на клавиатуре нет (даже в Лас-Вегасе). Конечно, такое обозначение было бы уместно, но в C оно не предусмотрено.

Зато дополнительные математические функции предусмотрены в обширной библиотеке языка C. Несколько десятков функций, предназначены для вычисления таких функций, как показательная (возведение в степень), квадратный корень, ужасающие тригонометрические функции и т.п. (см. табл. 25.1 далее в этой главе). Чтобы возвести число в степень, вызовите функцию `pow()`.

Функция `pow()` позволяет вычислить результат возведения значения в некоторую степень, например 4 во второй степени (4^2). Вот — формат:

```
value = pow(n,p)
```

В этой строке все переменные (`value`, `n` и `p`) должны быть переменными с удвоенной точностью, т.е. типа `double`. Иными словами, в определении (объявлении) этих переменных должен быть указан тип `double` (с удвоенной точностью). Функция `pow()` вычисляет `n` в степени `p`. Ответ сохраняется в переменной `value`.

Чтобы компилятор не сошел с ума, в начале исходного текста нужно включить заголовочный файл `MATH.H`. Вставьте следующую строку где-нибудь поближе к началу программы:

```
#include <math.h>
```

Эта строка требуется для функции `pow`, как и для других математических функций.

Использование функции pow()

Вызвать функцию `pow()` не сложнее, чем любую другую. Нужно только объявить ваши переменные с удвоенной точностью (тип `double`) и включить библиотеку `MATH.H` — все готово. Давайте рассмотрим следующую задачу.

Предположим, что вы с вашим другом Милтоном решили купить 2^8 (“два в восьмой степени”) новогодних огоньков-фонариков. Сколько огоньков-фонариков нужно купить?

Вы быстро садитесь за компьютер и на языке C пишете следующую программу:

```
#include <stdio.h>
#include <math.h>
int main()                                // главная функция
{
    double lights;                        // огни

    lights=pow(2,8); /* число 2 в 8-ой степени */
    // ("Милтон, нам нужно %0.f огней.\n", огни)
    printf("Milton, we need %0.f lights.\n",lights);
    return(0);
}
```

Напечатайте эту глупую программу на чистом экране в вашем редакторе. Сохраните это хитрое изобретение на диске под именем `LIGHTS1.C`.

Скомпилируйте и выполните.

Если в операционной системе семейства Unix компоновщик выдаст ошибку, пожалуйста, обратитесь к врезке “Компоновщик должен связать программу с математической библиотекой!”.

Вот вывод:

Milton, we need 256 lights.

(Милтон, нам нужно 256 огней.)

Теперь вы знаете, сколько необходимо огней. Компьютер вычислил, что 2^8 равно 256.



- ✓ Два в восьмой степени равно:
 $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2$
- ✓ Аргументы функции `pow` должны быть переменными типа `double` (с двойной точностью). И в самом деле, аргументы многих функций библиотеки `MATH.H` не могут быть целыми числами.
- ✓ Такие значения-константы, как 2 и 8 в `LIGHTS1.C`, не обязательно записывать как числа типа `double` (с двойной точностью). К счастью, компилятор довольно хитер и сделает все сам.
- ✓ Для печати числа с плавающей запятой в `printf()` используется метка-заполнитель `%f`. Она годится для чисел типа `float` и `double`. В `%0.f` часть `0.` указывает, что `printf()` форматирует выводимое число с плавающей точкой так, что печатается только часть числа, стоящая слева от десятичной точки.
- ✓ В некоторых языках и электронных таблицах для возведения в степень используется символ `“^”`. Например, 2^8 обозначает 2^8 . Но в C символ `“^”` для возведения в степень не применяется. (В C `^` обозначает логическую операцию поразрядного исключительного ИЛИ.)



Компоновщик должен связать программу с математической библиотекой!

Если вы используете Unix, Linux или Mac OS X, то обычно GCC конфигурируется так, что применять математические функции высокого уровня, такие как `pow()` и `sqrt()`, без дополнительных ухищрений не удастся. Причина в том, что стандартная библиотека для C не содержит таких математических функций. Преимущество: программы, скомпонованные со стандартной библиотекой, меньше. Недостаток: вы должны не забыть связать программу с математической библиотекой, если в программе используются некоторые математические функции.

Стандартная математическая библиотека в C называется `libm`. Чтобы связать ее с вашей программой, измените команду `gcc`:

```
gcc -lm source.c -o output
```

Обратите внимание на переключатель `-lm` в команде `gcc`. Этот переключатель указывает, что GCC должен связать программу с математической библиотекой `libm`, а не только со стандартной библиотекой. Входной файл — `source.c`, а выходной файл — `output`. Должным образом скомпилируйте `lights1.c`:

```
gcc -lm lights1.c -o lights1
```

Еще раз напомним, что эта опция необходима только для программ, в которых используются некоторые математические функции. Используйте команду `man`, чтобы найти руководство по нужной функции. Например, если хотите почитать руководство по использованию квадратного корня, введите команду

```
man sqrt
```

Переключатель `-lm` указывает, что стандартная математическая библиотека должна быть связана с программой, содержащей некую функцию. Стандартная математическая библиотека находится в каталоге `LIBRARY` (БИБЛИОТЕКА).

Извлечение корня

Вы, конечно, знаете, что есть математическая операция, обратная к возведению в квадрат, — извлечение квадратного корня. Я не могу привести практический пример, потому что, честно говоря, я считаю всю математику собранием глупостей. В C для этой операции символ на клавиатуре не предусмотрен. Чтобы извлечь квадратный корень из числа, используется специальная функция (как и для возведения в степень). В случае извлечения квадратного корня она называется `sqrt()`.

Функция `sqrt()` извлекает квадратный корень из числа, т.е. находит число, которое, умноженное на само себя, равняется тому числу, из которого вы хотите извлечь квадратный корень. Запомните, это функция `sqrt()`, а не `squirt` (шприц, струйка). Вот формат ее вызова:

```
value = sqrt(n)
```

Переменная `value` типа `double` равна квадратному корню из значения переменной `n` (также типа `double`). Да, здесь все должно быть типа `double`. Кроме того, в начале вашего исходного текста нужна следующая директива `include`:

```
#include <math.h>
```

Единственное ограничение на функцию `sqrt()` состоит в том, что число `n`, из которого вы хотите извлечь корень, не может быть отрицательным. Даже несмотря на то, что мои друзья-математики говорят мне, что есть такая вещь, как поле комплексных чисел, я подозрительно отношусь ко всяким попыткам извлечения квадратных корней из отрицательных чисел в программах на языке C.

А теперь давайте удивим вашего друга Милтона, вычислив квадратный корень из числа огней, купленных на Рождество. Чтобы упростить вычисление, воспользуемся программой `LIGHTS2.C`.

Напечатайте эту программу с помощью вашего редактора. Будьте внимательны при печати, потому что в функции `printf()` используется очень длинная строка. Чтобы показать, что эта строка продолжается на следующей строке, используется отдельная наклонная черта влево:

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

#define TOOTH 253

int main()                                // главная функция
{
    double lights;                        // огни

    lights=sqrt(256);
    /* Квадратный Корень из 256 */
    // Милтон, в результате получилось %0.f огней
    printf("Milton, I got your %0.f%c lights.\n",\
           lights,TOOTH);
    return(0);
}
```

Сохраните эту программу на диске под именем LIGHTS2.C.

Скомпилируйте и выполните. (Пользователи Unix, за подробностями обратитесь к врезке “Компоновщик должен связать программу с математической библиотекой!”.)

Milton, I got your 16² lights.

(Милтон, я получил ваши 16² огней.)

Функция `sqrt()` хвастается, что она знает, что квадратный корень из 256 равен 16. Или, выражаясь иначе, 16² (или 16×16) равно 256.

- ✓ Квадратный корень из 256 равен 16, а 16 в квадрате (16²) равно 256.
- ✓ Символ, имеющий код 253, представляет собой крошечную цифру 2 — обозначение квадрата числа.
- ✓ Символ, имеющий код 251, представляет собой традиционный символ квадратного корня. Даже если этот символ предусмотрен в вашем текстовом редакторе, для извлечения квадратного корня все равно нужна функция `sqrt()`.
- ✓ В некоторых случаях коды 253 и 251 отображают символы, отличающиеся от возведения в квадрат и квадратного корня.
- ✓ `%c` в строке формата функции `printf()` позволяет ввести специальный символ, 253, определенный как `TOOTH` ранее в исходном тексте.
- ✓ Однажды читатель по электронной почте прислал мне письмо, в котором спрашивал, есть ли в языке C некоторый эквивалент того, что в математике обычно обозначается через i . Напомню, что i представляет мнимое число $\sqrt{-1}$, или квадратный корень из “минус единицы”. Поскольку я знаю не все, я вынужден был сказать, что я не знаю. Конечно, в некоторых математических библиотеках языка C могут быть предусмотрены комплексные числа, но я об этом понятия не имею, хотя я полагаю, что вычисления с комплексными числами предусмотрены в языке программирования C++. (Однако я не работаю с C++, так что я не могу подтвердить это.)



Странная математика!

Большинство библиотек языка C просто напичканы математическими функциями. Их множество. Некоторые из наиболее часто встречающихся перечислены в табл. 25.1 вместе с их форматами. Подавляющее большинство из них используют в качестве аргументов значения типа `double` или `float`, которые служат для представления десятичных чисел.

Таблица 25.1. Математические функции

Функция	Название и что вычисляет	Формат	Включаемый файл	Библиотека
<code>abs</code>	абсолютное значение	<code>a=abs(b)</code>	<code>STDLIB.H</code>	стандартная (standard)
<code>acos</code>	арккосинус <code>arccos</code>	<code>x=acos(y)</code>	<code>MATH.H</code>	<code>libm</code>
<code>asin</code>	арксинус <code>arcsin</code>	<code>x=asin(y)</code>	<code>MATH.H</code>	<code>libm</code>
<code>atan</code>	арктангенс <code>arctg</code>	<code>x=atan(y)</code>	<code>MATH.H</code>	<code>libm</code>
<code>cos</code>	косинус	<code>x=cos(y)</code>	<code>MATH.H</code>	<code>libm</code>
<code>exp</code>	показательная функция	<code>x=exp(y)</code>	<code>MATH.H</code>	<code>libm</code>
<code>log</code>	натуральный логарифм	<code>x=log(y)</code>	<code>MATH.H</code>	<code>libm</code>
<code>log10</code>	десятичный логарифм, или логарифм по основанию 10	<code>x=log10(y)</code>	<code>MATH.H</code>	<code>libm</code>
<code>sin</code>	синус	<code>x=sin(y)</code>	<code>MATH.H</code>	<code>libm</code>
<code>tan</code>	тангенс <code>tg</code>	<code>x=tan(y)</code>	<code>MATH.H</code>	<code>libm</code>

- ✓ В табл. 25.1 переменные `a`, `b` и `c` обозначают целочисленные значения. Переменные `x`, `y` и `z` обозначают значения с удвоенной точностью (типа `double`).
- ✓ Библиотека `libm` необходима только в том случае, если компиляция программы выполняется в операционной системе семейства Unix. Обратитесь к врезке “Компоновщик должен связать программу с математической библиотекой!”
- ✓ Абсолютное значение числа — его значение без знака “минус”.
- ✓ Существуют функции даже еще более сложные, чем приведенные в табл. 25.1. Мудрые академики действительно балдеют от них.
- ✓ Что эти команды делают и как они работают, лучше всего выяснить в справочном руководстве по библиотеке языка C, которое поставляется вместе с вашим компилятором. Если вам повезет, все это находится в книге-справочнике; в противном случае вся эта информация находится в интерактивной справочной системе вашего компилятора.
- ✓ В большинстве случаев вместо переменных в круглых скобках математической функции могут быть постоянные значения. Например:
`x=sqrt(1024);`



Действительно странные конструкции: декремент и инкремент

Мы снова возвращаемся к операторам приращения ++ и уменьшения (декремента) --. Что и говорить, они — фавориты многих программистов, но могут серьезно озадачить! Например, рассмотрим следующий фрагмент кода:

```
b=a++;
```

Переменная *b* равняется содержимому переменной *a* плюс 1. Да или нет?

Помните, что само выражение *a++* может рассматриваться как инструкция. Если так, разве не она выполняется сначала? А раз так, *b* равняется *a+1*, или же все-таки *b* равняется *a*, а затем *a* увеличивается, т.е. *a* увеличивается после того, как будет выполнено присваивание?

Это вопрос, который имеет ответ — важный ответ, и вы его должны знать, если не хотите, чтобы ваши программы вели себя непредсказуемо.

Опасности использования ++

Запомните следующее важное правило.

Переменная увеличивается после ее использования, если ++ стоит после нее.

Пусть имеем инструкцию

```
b=a++;
```

Когда компилятор видит эту инструкцию, он немедленно записывает значение переменной *a* в переменную *b*. Затем значение *a* увеличивается. Чтобы освоиться с этим правилом, изучите программу INCODD.C.

Напечатайте эту программу с помощью вашего редактора. Сохраните ее на диске под именем INCODD.C.

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

```
int main()                                // главная функция
{
    int a,b;

    a=10;
    b=0;
    // перед приращением
    printf("A=%d and B=%d before incrementing.\n",a,b);
    b=a++;
    // после приращения
    printf("A=%d and B=%d after incrementing.\n",a,b);
    return(0);
}
```

Скомпилируйте программу и запустите на выполнение! Вот что получится на выводе:

A=10 and B=0 before incrementing.

A=11 and B=10 after incrementing.

(A=10 и B=0 перед приращением.

A=11 и B=10 после приращения.)

Первая строка означает следующее: переменным *a* и *b* были присвоены указанные значения. Вторая строка раскрывает загадку ++. Сначала *b* получает значение *a*, которое равно 10; затем это значение увеличивается и становится равным 11.



- ✓ Всякий раз, когда в инструкции `C` после переменной следует `++`, значение переменной увеличивается в последнюю очередь, после выполнения всех математических действий и присваиваний (обозначаемых знаком `=`) в этой строке.
- ✓ Это правило легко запомнить. Обратите внимание, что `++` стоит после переменной. Поэтому компилятор сначала видит переменную, берет ее содержимое, и лишь затем, как бы между прочим, увеличивает ее значение.

- ✓ Это правило применяется и в том случае, если `++` используется внутри более сложной конструкции, например, внутри условного оператора. Рассмотрим конструкцию

```
if (a=b++) // если (a=b++)
```

Здесь сначала вычисляется выражение внутри скобок и определяется истинность условия, и лишь затем увеличивается значение `b`.

- ✓ Рассмотрим теперь конструкцию, в которой сравниваются два значения

```
if (a==b++) // если (a==b++)
```

В этой конструкции нет ничего необычного, просто `b++` указано в операции сравнения, а не в предыдущей (или последующей) строке. Нужно, конечно, иметь в виду, что `b` увеличивается после того, как команда `if` сравнит значения. Если результатом операции сравнения будет `true`, `b` все равно будет увеличено. Вспомните об этом правиле, если увидите, что ваша программа ведет себя подозрительно.

То же самое относится и к `--`

Оператор уменьшения `--`, стоящий после переменной, также изменяет значение переменной после выполнения всех математических действий и присваиваний (обозначаемых знаком `=`) в этой строке. Чтобы убедиться в этом, рассмотрим программу `DECODD.C`:

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

```
int main() // главная функция
{
    int a,b;

    a=10;
    b=0;
    // перед уменьшением
    printf("A=%d and B=%d before decrementing.\n",a,b);
    b=a--;
    // после уменьшения
    printf("A=%d and B=%d after decrementing.\n",a,b);
    return(0);
}
```

Сохраните файл на диске под именем `DECODD.C`. Скомпилируйте и выполните его. Распечатка результатов доказывает, что уменьшение выполняется после выполнения всех математических действий и присваиваний:

```
A=10 and B=0 before decrementing.
A=9 and B=10 after decrementing.
```

(A=10 и B=0 перед уменьшением.

A=9 и B=10 после уменьшения.)

Значение b равно 10, что означает, что переменная была уменьшена после выполнения всех математических действий и присваиваний.



Все это имеет большое значение, и к настоящему времени вы должны хорошо знать, что происходит при выполнении программы. Если вы не можете запомнить это важное правило, записывайте приращения и уменьшения на отдельных строках, как показано в следующем примере:

```
a++;  
b=a;
```

Странное явление отражения ++ от переменной: ++a

Загрузите программу INCODD.C в ваш редактор. Измените строку

```
b=a+++;
```

так:

```
b=++a;
```

Эта строка выглядит довольно странно. Но это не совсем так. Вы можете подумать, что `++` — некоторая сверхъестественная команда языка C. Чтобы избежать такой крамольной мысли, напечатайте эту строку так:

```
b = ++a;
```

Здесь `++` — это все равно оператор приращения. Он все равно увеличивает значение переменной a. Однако поскольку он стоит перед переменной, сначала выполняется приращение. Эта конструкция означает “сначала увеличить, а затем сделать то, что записано в инструкции”.

Сохраните исходный текст INCODD.C на диске. Скомпилируйте и выполните его. Вот что вы увидите:

```
A=10 and B=0 before incrementing.
```

```
A=11 and B=11 after incrementing.
```

(A=10 и B=0 перед приращением.

A=11 и B=11 после приращения.)

Это сработало! Сначала была выполнена операция `++`.

В этой программе вы встретили кое-что странное, о чем вам следует знать. Редко встречал я оператор `++` перед переменной, но иногда он может там встретиться. И эта конструкция может помочь сгладить некоторые странные вещи, если неприятности возникают из-за того, что вы помещаете `++` после имени переменной.

- ✓ Вы можете использовать `++` до или после переменной, если этот оператор появляется в отдельной строке. Если никаких других операций нет, нет и разницы, где поставить `++` — до или после переменной. Команда

```
a+++;
```

делает то же самое, что и команда

```
++a;
```

- ✓ Запись `a++` иногда называется *постинкрементом* или *постприращением*. Запись `++a` иногда называется *преинкрементом* или *предприращением*.

- ✓ Да, то же самое относится и к уменьшению. Вы можете изменить строку `b=a--;` в DECODD.C так:

```
b=--a;
```

Сохраните, скомпилируйте и выполните программу. Переменная a сначала уменьшается, а затем ее новое значение присваивается переменной b.



✓ Даже не беспокойтесь об этом:

```
++a++;
```

Вы можете подумать, что эта конструкция сначала выполняет приращение переменной, а затем увеличивает ее снова. Неправильно! Компилятор думает, что вы забыли что-нибудь, и когда вы попытаетесь скомпилировать такую инструкцию, вы получите ужасное сообщение об ошибке, что-нибудь вроде `Lvalue required` (требуется именующее выражение (адрес переменной)). Это сообщение означает, что компилятор где-нибудь поблизости ожидал встретить еще одну переменную или число. К сожалению, такая конструкция недопустима.

Старая функция генерации случайных чисел

В этой главе...

- Знакомство с функцией `rand()`
- Засев (инициализация) функции `rand()`
- Выполнение вычислений по модулю (%)
- Генерация случайных чисел в заданном диапазоне

В соответствии с первоначальным планом эта книга должна была содержать приблизительно 1 600 страниц. Я не хотел нарушать медленный, последовательный темп преподавания языка С. Проблема состоит в том, что невозможно втиснуть 1 600 страниц в книгу с 400 страницами, ничего не сокращая. Что же сократить, чем пожертвовать?

После всего того, что я выбросил, вы можете считать странным мое желание сохранить данную главу, посвященную функции генерации случайных чисел. Я сделал это потому, что, несмотря ни на что, случайные числа играют важную роль в программировании компьютера. Они незаменимы, например, в машинных играх. Ни одна машинная игра не была бы возможна без генерации случайных чисел. Без них игра была бы предсказуема: “О, я снова обхожу планету Бозар, и именно здесь Всеядная Пространственная Стрела пробует затянуть меня в этот водоворот времени”.

Случайность

Что является случайным? Карта, вытянутая из хорошо перетасованной колоды. Выпавшая грань игральной кости. Вращение колеса рулетки. Разрешит ли ваша жена вам выпить огромную кружку пива в ваш первый день после развода. Все эти события случайны, ибо они могут произойти или не произойти. Если вы хотите включить случайные события в ваши программы, вам нужна специальная функция, которая генерирует случайные числа. После того как вы включите случайность, никто не сможет предсказать, когда будет атаковано Предприятие или когда торнадо захлестнет вашу деревню, выведет ли на сей раз Дверь номер 2 к сокровищу или к некоторой опасности.

- ✓ Телефонная компания, по слухам, имеет программу генерации случайных чисел, которую она обычно использует для заполнения счетов.
- ✓ Подпрограммы генерации случайных чисел — самое сложное испытание для подающего надежды программиста. Я постараюсь избавить вас от их разработки в этой книге. (Ну, возможно, не полностью.)
- ✓ В языке С случайные числа генерирует функция `rand()`.



- ✓ Программу генерации случайных чисел нужно инициализировать, чтобы числа были непредсказуемыми. (Далее в этой главе я рассматриваю также и эту тему.)
- ✓ Случайные числа, генерируемые компьютером, на самом деле не случайны. Подробности приведены во врезке “Вы можете впустую потратить несколько секунд, читая эту информацию о случайных числах”.



Вы можете впустую потратить несколько секунд, читая эту информацию о случайных числах

Действительно ли числа, генерируемые программами, являются случайными? Только в том случае, если они не могут быть предсказаны. К сожалению, числа, генерируемые компьютерами, могут быть предсказаны. Они более или менее неожиданны, подобно номерам улиц в Сياتле, но в целом случайные числа, которые генерирует компьютер, на самом деле не случайны. Вместо этого они псевдослучайны.

Псевдослучайное число достаточно случайно для большинства целей. Но поскольку число вычисляется по определенному алгоритму компьютером или определенной подпрограммой, результат в действительности не случаен. Даже если в вычислении случайного числа используется время дня или генератор случайных чисел инициализируется другим потенциально случайным значением, результат все же не настолько случаен, чтобы успокоить математических пуритан. Так что с этим нужно мириться.

Функция rand()

В С случайные числа генерирует функция `rand()`. Она возвращает случайное число, зависящее от прихотей микропроцессора вашего ПК, даты рождения программиста, который написал ваш компилятор с языка С, веса его очередной подружки, а также еще от нескольких подобных параметров.

Вот формат функции `rand()`:

```
int rand();
```

Функция `rand()` возвращает целочисленное значение, обычно в диапазоне от 0 до 32 767. Чтобы сохранить случайное число в переменной `r`, используется следующая инструкция:

```
r=rand();
```

Чтобы компилятор знал о функции `rand()`, в начале вашего исходного текста нужно добавить следующую строку:

```
#include <stdlib.h>
```

Эта строка содержит все подробности о функции `rand()`, нужные компилятору, и делает его (и программиста) счастливым.

Следующая программа показывает функцию `rand()` в действии. `RANDOM1.C` генерирует 100 случайных чисел, которые она аккуратно отображает на экране. В этой программе случайные числа генерируются функцией генерации случайных чисел — функцией `rnd()`. Это было сделано для того, чтобы было легко изменять функцию `rnd()` в процессе чтения данной главы.

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

```
int rnd(void);
```

```
int main()
{
```

```
// главная функция
```

```

int x;
// отобразить ("Смотрите! 100 Случайных чисел! ");
puts("Behold! 100 Random Numbers!");
for(x=0;x<100;x++)
    printf("%d\t",rnd());
return(0);
}

int rnd(void)
{
    int r;

    r=rand();
    return(r);
}

```

Напечатайте исходный текст RANDOM1.C с помощью вашего редактора. Перепроверьте все. Сохраните файл на диске под именем RANDOM1.C.

Скомпилируйте программу. Исправьте все ошибки. (Вероятнее всего, они связаны с отсутствием точек с запятой или с забытыми круглыми скобками.)

Выполните программу!

На устройстве вывода (обычный пультовой дисплей) отображается десять столбцов и десять строк случайных чисел — нет смысла повторять их здесь! Но обратите внимание, что на вашем экране числа случайны. А вы смогли бы так быстро назвать столько случайных чисел?

- ✓ Вероятнее всего, случайные числа находятся в диапазоне от 0 до 32 000 — как и было обещано. Числа, которые вы видите на вашем экране, вероятно, отличаются от тех чисел, которые будут отображены на другом компьютере.
- ✓ Некоторые версии компилятора GCC генерируют значения случайных чисел не в диапазоне от 0 до 32 000, а в большем. Максимальное значение равно значению RAND_MAX, которое определено в заголовочном файле SDTLIB.H. На моей машине с FreeBSD, например, значение RAND_MAX определено как 0x7FFFFFFF (шестнадцатеричное), что равно 2 147 483 647.
- ✓ Чтобы увидеть значение RAND_MAX на вашем компьютере, добавьте к программе в главной функции main() следующую строку перед return(0);:


```
printf("\nRAND_MAX is equal to %u\n",RAND_MAX);
```

 Здесь %u — метка-заполнитель для длинного целого числа без знака (unsigned long).

Инициализация генератора случайных чисел

Прежде чем приступить к новой программе CASINO, давайте запишем (и продадим за миллионы в Лас-Вегас!) сто случайных чисел. Для этого снова сделаем повторный запуск программы RANDOM1.C. Выберите Run в интегрированной среде вашего компилятора или напечатайте команду RANDOM1 в подсказке DOS.

(Вывод я опустил!)

Да, на экране вы видите очередные 100 случайных чисел. Подождите секунду. Разве это не те же самые числа? *Те же самые числа?* Компилятор попал впросак? Пытался обмануть?

К счастью, это не ошибка. Компьютер просто генерирует набор случайных чисел. Несомненно, они случайны, но они — те же самые числа, потому что, запуская функцию rand(), компилятор не изменяет ее.

Не чувствуйте себя обманутым! Это обычная ситуация. Функция `rand()` только квази-случайна. Чтобы делать ее более случайной (более псевдослучайной), вы должны инициализировать ее начальным значением. Это начальное значение (число) использует компилятор, чтобы помочь ей сделать случайные числа более случайными. Чтобы установить начальное число, используется функция `srand()`.

Функция `srand` помогает инициализировать компьютерный механизм генерации случайных чисел более случайным способом. Вот формат:

```
void srand((unsigned) seed)
```

Значение начального числа `seed` — целое число без знака (`unsigned`) или значение переменной в диапазоне от 0 до (примерно) 65 000. Это то значение, которое помогает компилятору настроить генератор случайных чисел вашего ПК.

В начале исходного текста для функции `srand()` нужна следующая строка:

```
#include <stdlib.h>
```

Поскольку функция `rand()` также требует этой строки, нет никакой необходимости включать ее дважды.



- ✓ Конструкция `(unsigned)` (без знака) гарантирует, что число, которое использует `srand()`, имеет тип `unsigned`, и потому не может быть отрицательным. Этот прием называется приведением типа.
- ✓ Используя значение 1 (один) для инициализации генератора случайных чисел, можно заставить компилятор начать генерацию случайных чисел заново — числа будут генерироваться так, как если бы `srand()` не использовалась. Избегайте такого применения `srand()`, если возможно.

Повышаем непредсказуемость программы RANDOM

Теперь пришло время сгенерировать действительно случайные числа. Следующий исходный текст — программа `RANDOM2.C`, умеренная модификация первоначальной программы `RANDOM`. На сей раз добавляется новая функция `seedrnd()`, которая позволяет сбрасывать генератор случайных чисел и генерировать более непредсказуемые последовательности случайных чисел:

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

int rnd(void);
void seedrnd(void);

int main()                                // главная функция
{
    int x;

    seedrnd();
    // "Смотрите! 100 Случайных чисел!"
    puts("Behold! 100 Random Numbers!");
    for(x=0; x<100; x++)
        printf("%d\t", rnd());
    return(0);
}

int rnd(void)
{
    int r;
```

```

r=rand();
    return(r);
}

/* инициализировать генератор случайных чисел */
void seedrnd(void)
{
    int seed;
    char s[6];

    // Введите начальное число для генератора случайных чисел (2 - 65000):
    printf("Enter a random number seed (2 - 65000):");
    seed=(unsigned)atoi(gets(s));
    srand(seed); // начальное число
}

```

Напечатайте эту программу с помощью вашего редактора. Вы можете начать редактировать исходный текст RANDOM1.C. Добавьте в его начало прототип `seedrnd()`, а затем вставьте вызов `seedrnd()` в главной функции `main()`. И последний штрих: в конце исходного текста добавьте саму функцию `seedrnd()`. Перепроверьте все это прежде, чем вы сохраните текст, чтобы удостовериться, что вы ничего не пропустили.

Используйте команду редактора Save As (Сохранить как), чтобы сохранить файл на диске под именем RANDOM2.C.

Скомпилируйте и выполните. Вы увидите следующую строку:

Enter a random number seed (2-65000):

(Введите начальное число для генератора случайных чисел (2-65000):)

Напечатайте число из диапазона от 0 примерно до 65 000. Нажмите <Enter>, и вы увидите новый и еще более случайный набор отображаемых чисел.

Чтобы по-настоящему испытать программу, выполните ее снова. На сей раз напечатайте другое начальное число. Следующая последовательность случайных чисел полностью отличается от первой.



- ✓ Генератор случайных чисел нужно инициализировать только однажды, как это делает программа в главной функции `main()`. Некоторые пуристы настаивают на многократном вызове функции `seedrnd()` (или ее эквивалента). Но случайное случайно настолько, насколько случайным может быть то, что делает компьютер. Нет смысла тратить время.

- ✓ В этой программе три инструкции C объединены в одну. Вот эта инструкция:
`seed=(unsigned)atoi(gets(s));`

Эта строка — объединение следующих двух инструкций:

```

gets(s);
seed=(unsigned)atoi(s);

```

Сначала `gets()` читает строковую переменную. Затем прочитанное значение преобразуется функцией `atoi` и, наконец, записывается в целую переменную `seed` — переменную, в которой хранится начальное число в надлежащем формате, т.е. без знака (`unsigned`).

- ✓ Функция `seedrnd()` не обязана возвращать какое-либо значение. В главе 22 “Как на самом деле функционируют функции” уже объяснялось, что некоторые функции не принимают и не возвращают значений. Эта функция — одна из них. Все же важно вызвать функцию, потому что она засеивает (инициализирует) генератор случайных чисел, используемый в программе в другом месте.

Упрощение генератора случайных чисел

Ничто не раздражает так, как неряшливая программа. Я говорю не о том, как она написана, а о том, как она выглядит — о ее представлении. Когда программа просит инициализировать генератор случайных чисел, это неправильно. Компьютер должен сделать это самостоятельно. И он может это сделать, если найдет источник постоянно изменяющихся чисел, которые он может использовать для инициализации генератора случайных чисел.

Один из таких источников — часы компьютера. Большинство ПК отсчитывает время с точностью до сотой доли секунды: каждую 1/100 секунды генерируется новое число, которое можно использовать для инициализации механизма генерации случайных чисел. Впрочем, для инициализации механизма генерации случайных чисел вполне подошло бы даже текущее время в секундах. Это было бы число, увеличивающееся только на 1 через каждую секунду.

Следующий исходный текст — программа RANDOM3.C. Эта программа почти идентична RANDOM2.C за исключением того, что функция `seedrnd()` теперь использует текущее время для инициализации генератора случайных чисел:

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

int rnd(void);
void seedrnd(void);

int main()                                     // главная функция
{
    int x;

    seedrnd();
    // "Смотрите! 100 Случайных чисел!"
    puts("Behold! 100 Random Numbers!");
    for(x=0; x<100; x++)
        printf("%d\t", rnd());
    return(0);
}

int rnd(void)
{
    int r;

    r=rand();
    return(r);
}

/* инициализировать генератор случайных чисел */
void seedrnd(void)
{
    srand((unsigned)time(NULL));
}
```

Создайте исходный текст RANDOM3.C в вашем редакторе. Вы можете начать с редактирования RANDOM2.C. Добавьте новую строку `#include <time.h>`, которая позволяет использовать часы компьютера для инициализации генератора случайных чисел. Затем измените функцию `seedrnd()` так, как показано в настоящей версии исходного кода. Сохраните файл на диске под именем RANDOM3.C с помощью команды Save As (Сохранить как) вашего редактора.

Скомпилируйте и выполните программу. Обратите внимание, что формат вывода не изменился, но числа совсем другие. Важно отметить, что программа не просила инициализировать генератор случайных чисел. На этот раз все было сделано автоматически.



- ✓ Функция `time()` в строке `srand((unsigned)time(NULL));` возвращает время дня, отсчитываемое внутренними часами компьютера. Функция вызывает-ся с постоянным значением `NULL` (ПУСТОЙ УКАЗАТЕЛЬ), которое на самом деле является значением 0. (Эта константа определена в заголовочном файле `stdio.h`, так что все делает строка `#include <stdio.h>`; подробности см. в главе 23 “То, что пишется в начале программы”).
- ✓ Значение, возвращаемое функцией `time()`, — некоторое число, притом не важно, какое именно. Важно лишь, что это значение имеет формат целого числа без знака (`unsigned`), — этот формат оно приобретает благодаря приведению типа (`unsigned`).
- ✓ В программе на С следующая инструкция случайным образом инициализирует генератор случайных чисел:

```
srand((unsigned)time(NULL));
```
- ✓ Для функции `time()` (а поэтому и для предыдущей инструкции инициализации генератора случайных чисел) требуется следующая инструкция в начале исходного текста:

```
#include <time.h>
```
- ✓ Кроме того, требуется, конечно, и вот эта инструкция:

```
#include <stdlib.h>
```

Дьявольский доктор Модуль

Вы, возможно, заметили, что случайные числа, сгенерированные в программах СЛУЧАЙНЫМ образом, были случайными до дикости и часто не подходили для решения конкретной задачи. Представьте, например, такое сообщение:

Ваша очередь, БОБ.

Вы выбросили 23415 на костях. Это переносит вас в Волшебный лес, но вы прошли этой дорогой 585 раз, и потому вы зарабатываете 117 000 \$!

Это немного озадачивает.

Нужно придумать способ, чтобы уменьшить числа так, чтобы значения (числа) всегда лежали только в некотором заданном диапазоне.

И способ такой есть. Он использует математическое понятие *модуля*.



Математическое понятие!!!

Модуль — вот еще! Этот термин вам уже знаком — это “остаток от деления”. Когда вы делите 10 на 6, вы получаете остаток 4. Следовательно, 4 — это 10 по модулю 6. Это можно записать на С так: `4 = 10 % 6;`

Я не предполагаю, что вы знаете, что такое модуль, и помните, когда нужно его использовать. Формат этого оператора тот же самый, что и у любого другого математического оператора в языке С. Важно знать, что модуль позволяет “срезать” большие числа, превращая их в меньшие, более удобные.

`остаток = большое число % модуль;`

Читайте это так. Если взять огромное, большое число и разделить его на меньшее (модуль), получится остаток, который сравним с большим значением по указанному модулю.

Предположим, что большое число встретилось в следующей инструкции:

```
m = большое число % 5;
```

Значения переменной *m* находятся в диапазоне от 0 до 4, потому что они равны остатку от деления большого числа на 5.

Значение переменной *m* после выполнения следующей инструкции равно 0 или 1, в зависимости от того, является ли *odddereven* четным или нечетным:

```
m = oddereven % 2;
```

Например, кость (в виде игрального кубика) имеет шесть граней. Предположим, что компьютер генерирует случайное значение 23 415. Чтобы найти остаток от деления этого числа на 6, запишите следующую строку:

```
dice1=23415 % 6;
```



Компьютер вычисляет остаток от деления 23 415 на 6. Затем он помещает остаток в переменную *dice1*. (Результат равен числу 3, которое более реалистично в качестве результата бросания кости, чем 23 415.)



- ✓ Если второе значение больше первого, как в `5 % 10`, результат всегда равен первому значению. Поэтому обычно большее значение является первым операндом операции вычисления остатка по модулю.
- ✓ Оператор вычисления остатка по модулю — `%`, знак процента. Он читается “остаток по модулю” или “остаток от деления”.
- ✓ Никаких сложных математических вычислений! Модуль помогает уменьшить случайные числа. Вот и все! Вы можете найти свойства математической операции `%` в других книгах по языку C.
- ✓ Если вы хотите большое случайное число *random_value* использовать для вычисления случайного числа, которое может выпасть на грани игральной кости, запишите инструкцию:

```
dice1=(random_value % 6)+1;
```

Генерируемое компьютером число *random_value* нужно уменьшить с помощью операции `% 6`. В результате этой операции получается число в диапазоне от 0 до 5 (остаток от деления на 6 находится в диапазоне от 0 до 5 — ведь остаток от деления 6 на 6 равен 0). После вычисления результата операции `%`, вы добавляете 1 к числу и получаете значение в диапазоне от 1 до 6, а ведь именно числа из этого диапазона как раз и написаны на всех гранях игральной кости.

- ✓ Вспомните теперь Мою Дорогую Тетушку Салли. Операция вычисления модуля выполняется только после деления и перед сложением. См. врезку “Знакомьтесь с родственницей по матушке: Моя Дорогая Мамина Тетушка Салли”.
- ✓ “Ах, да, доктор Модуль. Я знаком с вашей работой по астрогенетике. Это правда, что вас выгнали с треском из академии за то, что вы принялись проектировать мышей третьего пола?” “Вы слишком много читаете сплетен”.



Знакомьтесь с родственницей по матушке: Моя Дорогая Мамина Тетушка Салли

Вы читали о Моей Дорогой Тетушке Салли в главе 11 "Больше математики и Священный порядок (старшинство операций)"? Это — мнемоническое устройство (вещь, которая помогает вам запомнить что-нибудь). Моя Дорогая Тетушка Салли (*My Dear Aunt Sally*) позволяет запомнить порядок выполнения математических действий в длинной инструкции на языке C: multiplication (умножение), division (деление), addition (сложение) и subtraction (вычитание). Добавьте теперь операцию вычисления остатка по модулю, которая имеет приоритет перед сложением и вычитанием. Вот порядок выполнения математических действий:

Моя Дорогая Мамина Тетушка Салли:

* Multiplication (умножение)

/ Division (деление)

% Modulus (остаток по модулю)

+ Addition (сложение)

- Subtraction (вычитание)

Поэтому следующая инструкция, в которой вычисляется количество очков, выпавших на грани игрового кубика:

```
dice1 = (23415 % 6) + 1;
```

делает то же самое, что и инструкция без круглых скобок:

```
dice1=23415 % 6+1;
```

Сначала выполняется операция вычисления остатка по модулю ($23415 \% 6$), а затем 1 добавляется к результату. Конечно, если вы вставили круглые скобки для удобочитаемости, компилятор ругаться не будет.

Окончательная версия программы *RANDOM*; СЛУЧАЙНЫМ образом бросаем игровой кубик

Если вы выполняли программы, описанные в этой главе, вы должны были получать последовательности случайных чисел, выпрыгивающих подальше от вашего компьютера, подобно жителям Нью-Йорка, покидающим свои дома при падениях напряжения в августе. Однако нужно сделать еще одно важное усовершенствование в функции `rnd()`. Пришло время добавить автоматическое средство поиска диапазона. Следующая программа, *RANDOM4.C* (последняя в этом наборе программ), имеет функцию `rnd()`, которая генерирует случайные значения только в диапазоне от 0 до любого наперед заданного числа. Эта программа должна внести некоторый порядок в ваши случайные числа:

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

```
int rnd(int range);
void seedrnd(void);
```

```
int main()
```

```
// главная функция
```

```

{
    int x;

    seedrnd();
    for(x=0;x<100;x++)
        printf("%i\t",rnd(10));
    return(0);
}

int rnd(int range)
{
    int r;

    r=rand()%range;
    return(r);
}

void seedrnd(void)
{
    srand((unsigned)time(NULL));
}

```

Создайте исходный текст RANDOM4.C. Начните с редактирования программы RANDOM3.C и внесите в исходный текст только что показанные модификации. Сохраните файл на диске под именем RANDOM4.C.

Скомпилируйте и выполните программу. Вот — выборка из вывода:

```

4   1   3   0   6   6   1   0   8   9
2   9   5   9   8   7   6   8   0   9
5   6   2   0   5   8   5   5   9   0
9   9   2   6   1   2   0   2   0   7
8   4   4   7   1   6   0   0   5   1
3   7   1   2   1   2   5   0   8   5
9   2   0   7   9   8   4   5   6   0
8   8   7   6   0   8   3   9   3   4
0   4   0   5   5   6   3   0   4   3
7   6   1   2   2   7   6   7   4   8

```

Все числа находятся в диапазоне от 0 до 9, поскольку этот диапазон задан в вызове rnd(10) в строке printf("%i\t",rnd(10));.

- ✓ Функции rnd() и seedrnd() удобно использовать при написании программ на С (обычно игр), но при необходимости можно копировать и вставлять эти функции в другие программы. Помните, что обе функции требуют директивы #include <stdlib>, а seedrnd() требует также #include <time.h>.
- ✓ Чтобы генерировать количество очков, выпавших на грани игрального кубика, вставьте в программу функцию rnd() и используйте следующую инструкцию:
dice=rnd(6)+1; /* Бросить игральный кубик! */
- ✓ Используя функциональный стиль в языке С, вызов функции rnd() можно записать в одной инструкции:
return(rand()%range);
- ✓ Теперь осталось изучить сущие пустяки, которых вам недостает для написания вашей собственной игры Монополия...



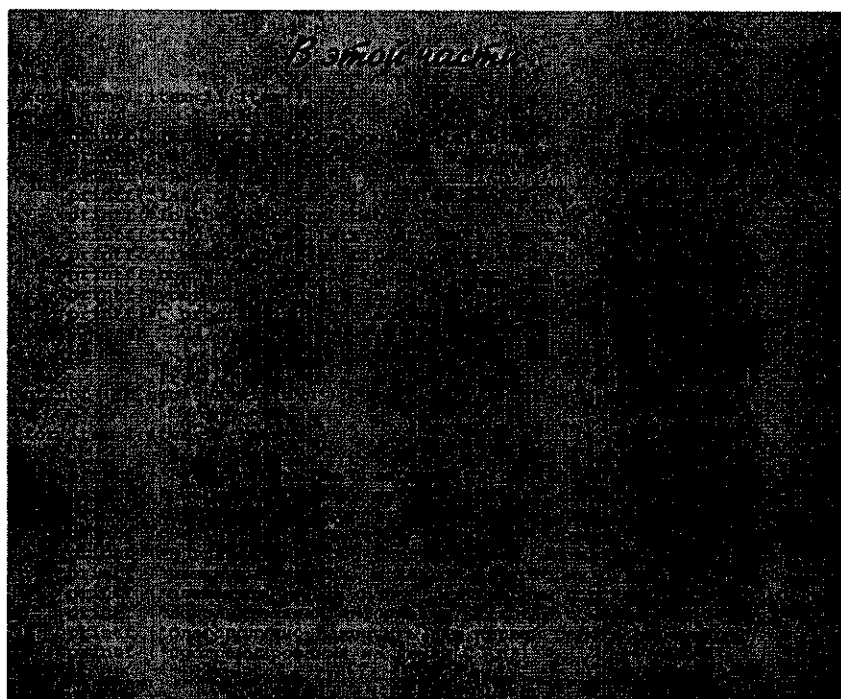
Часть V

Великолепные десятки

Настоящие программисты



Настоящие программисты лишь всего работают между часом
ночи и пятью утра



Еще десять трюков в языке C

В этой главе...

- Массивы
- Строки
- Структуры
- Указатели
- Связанные списки
- Побитовые операции
- Использование параметров командной строки
- Доступ к файлам на дисках
- Взаимодействие с операционной системой
- Разработка больших программ

Ваша поездка в страну программирования на языке C еще очень далека от окончания. Язык C настолько обширен, что он не мог быть полностью описан в этой книге, особенно если учесть наш неторопливый темп.

Чтобы дать вам представление о других темах, я включил сюда десять наиболее важных понятий языка C — да и программирования вообще.

Массивы

Массивы имеют множество вариантов. В массиве можно хранить много различных целых чисел, значений с плавающей точкой, символы, строки или значения переменных любого типа — и все это в одной-единственной переменной. Предположим, что 32, 17, 96 — ключевая комбинация к Большому Сейфу в Форт-Ноксе.

Эти три числа — целые числа, конечно. Но все три, хранимые вместе, образуют массив.

- ✓ Массивы объявляются точно так же, как другие переменные, хотя после имени переменной следуют квадратные скобки:

```
int combination[3];
```

В этом примере объявлен массив `combination` (комбинация). Он может содержать три элемента (в определении на это указывает число 3 в квадратных скобках).

- ✓ В одной инструкции можно объявить массив и инициализировать его:

```
float temps[] = { 97.0, 98.2, 98.6, 99.1 };
```

Этот массив называется `temps` и содержит четыре значения с плавающей точкой.

- ✓ Можно обратиться к каждому отдельному элементу массива. Массив состоит из элементов.



- ✓ Первый элемент в массиве занумерован числом нуль. Это важно помнить. Так, нулевой элемент массива `temps[]` равен `97.0`.
- ✓ Значения элементам массива присваиваются так же, как и обычным переменным. Например:

```
combination[0] = 32;  
combination[1] = 17;  
combination[2] = 96;
```

Эти три инструкции присваивают значения 32, 17 и 96 трем элементам массива `combination[]`). (Обратите внимание, что индекс первого элемента равен нулю.)
- ✓ В языке С необходимо установить размер массива. После того как установлен размер и количество измерений (размерность), массив может содержать определенное количество элементов, причем это количество устанавливается один раз и не может быть изменено в будущем ни при каких обстоятельствах. Досадно, но ничего поделать с этим нельзя.

Строки

Строка — не что иное, как массив символов. Например:

```
char myname[] = "Dan";
```

Это объявление создает строковую переменную, названную `myname`. Содержимое этой переменной — `Dan` (Дэн), или символы `D`, `a` и `n`. Вы можете также записать это в более традиционном для массива стиле:

```
char myname[] = {'D', 'a', 'n'};
```

Строки всегда заканчиваются **НУЛЕВЫМ СИМВОЛОМ NULL** с ASCII-кодом 0, который определен в `STDIO.H` как слово `NULL` (ПУСТОЙ УКАЗАТЕЛЬ) или символьный код `\0` (нуль с наклонной чертой влево). Вот — длинный способ создать строку `myname`:

```
myname[0] = 'D';  
myname[1] = 'a';  
myname[2] = 'n';  
myname[3] = '\0';
```

Заключительный **НУЛЕВОЙ СИМВОЛ NULL** указывает компилятору, где заканчивается строка. Ведь строки могут содержать символы вроде `<Enter>` и `<Tab>` (позиция табуляции). В языке С для обозначения конца текста используется символьный код 0, или `NULL`.

- ✓ Строка — не что иное, как символьный массив.
- ✓ Все строки заканчиваются **НУЛЕВЫМ СИМВОЛОМ NULL**.
- ✓ В этой книге для чтения строк используются функции `scanf()` и `gets()`. В функции `printf()` для отображения значения строки можно использовать метку-заполнитель `%s`.
- ✓ В языке С для обработки строк предусмотрено очень много функций.
- ✓ При обработке строк в языке С часто используется функция `strcmp()`, которая сравнивает строки. Помните, что в условном операторе нельзя сравнивать строки с помощью `==`, но вы можете использовать `strcmp()` или подобные функции.



Структуры

Язык С позволяет объединять переменные в единый пакет, называемый структурой. Такой пакет подобен записи в базе данных, поскольку переменная-структура может хранить множество различных данных. Так что это некий эквивалент карточки 345.

Структуры объявляются с помощью ключевого слова `struct`. После него следует название (имя) структуры и ее содержимое:

```
struct sample                                // выборка
{
    int a;
    char b;
}
```

Эта структура называется `sample` (выборка). Она содержит две переменные: целое число, названное `a`, и символ, названный `b`. Обратите внимание, что эта команда просто создает структуру, она не объявляет никаких переменных. Чтобы сделать это, нужна еще одна строка.

Следующая строка объявляет переменную-структуру, названную `s1`. Структура, которую использует эта переменная, имеет тип, определенный как `sample` (выборка):

```
struct sample s1;
```

Предположим, что вы пишете игру и вам нужен некоторый способ отслеживать символы в игре. Рассмотрим следующую структуру:

```
struct character                             // СИМВОЛ
{
    char name[10];
    long score;                               // счет
    int strength;                             // сила
    int x_pos;
    int y_pos;
}
```

Эта структура называется `character` (символ). Она содержит переменные, которые описывают переменные атрибуты символа в игре: `name` (название символа), `score` (счет), `strength` (сила) и местоположение на игровой сетке (`x_pos` и `y_pos`).

Чтобы определить четыре символа, используемые в игре, необходимы следующие объявления:

```
struct character g1;
struct character g2;
struct character g3;
struct character g4;
```

Впрочем, их можно заменить одним:

```
struct character g1, g2, g3, g4;
```

Чтобы обратиться к элементу структуры, используется нотация с точкой. Вот как отображается название символа (`name`) для `g1`:

```
printf("Character 1 is %s\n", g1.name);
```

Предположим, что символ `g2` подкошен ударом меча:

```
g2.strength -= 10;
```

Эта инструкция вычитает 10 из значения `g2.strength`, т.е. из целого числа, представляющего силу (`strength`) в структуре-символе `g2` типа `character`.

- ✓ Да, структуры в языке С подобны записям в базах данных.
- ✓ Очевидно, структуры — это обширная тема, ее нельзя охватить в одном разделе.

Указатели

В главе 1 “Основы языка С” говорилось о том, что язык программирования С — язык среднего уровня, потому что ему присущи черты языка программирования как нижнего уровня, так и высокого уровня. А теперь... Добро пожаловать на самый низкий уровень. Указатели используются в С, чтобы непосредственно управлять памятью компьютера — в частности, хранением переменных в памяти.

На первый взгляд эта идея кажется совершенно бесполезной. Действительно, зачем нужен указатель, когда вы можете изменить значение переменной, используя функцию или знак “=”? И все же указатели в языке С создают некоторые дополнительные возможности, отличающие его от любого другого языка программирования.

Проблема с указателями состоит в том, что для того, чтобы понять их, требуется четыре или пять раз прочитать посвященный им раздел, а затем тщательно разобрать шесть или семь программ, показывающих, как они могут использоваться. Прежде чем вы действительно освоите их, придется изрядно потрудиться.

- ✓ Указатели объявляются с помощью звездочки, которая сама по себе может запутать, потому что звездочка обозначает также умножение.
- ✓ Следующая строка объявляет целочисленный указатель `s`:
`int *s;`
- ✓ Рассказывая об указателях, пришлось бы рассказать также об амперсандах, но они используются настолько причудливо, что я не осмеливаюсь далее упоминать о них.
- ✓ Главное правило применения указателей: не присваивайте им случайных значений. Все указатели должны быть объявлены, им нужно присвоить значения, и только затем их можно использовать.

Связные списки

Хотите ужаснуть любого второкурсника университета? Упомяните связанные списки, и он побледнеет от ужаса. Связные списки — важное средство языка С, почти столь же неприятное, как и указатели. Но в действительности они не представляют собой ничего ужасного, это только немного странное понятие. Странное, поскольку для того чтобы разобраться в связанных списках, вы должны освоить указатели. Это может быть трудновато.

В языке С связанные списки объединяют понятие массива со структурами и указателями. В некотором смысле связный список действительно походит на массив структур. Однако в отличие от массива структур, в список связей можно легко добавлять структуры и также легко эти структуры удалять из него.

И это все, что я действительно хотел сказать о связанных списках!

Двоичные операторы

Наряду с указателями, к средствам нижнего уровня языка С относятся также двоичные операторы. Двоичные операторы позволяют выполнять операции над битами в любой переменной языка С, кроме переменных типа `float` (с плавающей точкой) и `double` (двойной точности). В табл. 27.1 перечислены двоичные операторы.

Таблица 27.1. Поразрядные операторы

Оператор	Функция
&	И (AND)
^	Исключительное ИЛИ (Exclusive OR (EOR или XOR) — неэквивалентность)
	Включительное ИЛИ (Inclusive OR)
~	Поразрядное дополнение
<<	Побитовый сдвиг влево
>>	Побитовый сдвиг вправо

Эти операторы используются только тогда, когда программы взаимодействуют с аппаратными средствами ПК или с операционной системой. Например, чтобы определить, установлен ли для некоторого порта ПК его младший бит, можно использовать один или несколько этих операторов.

Особенность операторов >> и << состоит в том, что они выполняют сверхбыстрые операции деления и умножения двоичных чисел. Например:

```
x = y >> 1;
```

Эта функция делит значение *y* на 2, поскольку она сдвигает биты в значении *y* на один разряд влево. Она выполняется намного быстрее, чем инструкция

```
x = y/2;
```

Чтобы разделить *y* на 4, вы можете использовать следующую функцию:

```
x = y >> 2;
```

Правда, вы должны разбираться в двоичной арифметике, но это вполне возможно. Аналогично оператор << позволяет удвоить значение, но об этом поговорим как-нибудь в другой раз.

Использование параметров командной строки

В главе 22 “Как на самом деле функционируют функции” вы узнали о том, как главная функция `main()` возвращает значение операционной системе, когда программа завершается. Это — один из способов, которым программа может связаться с операционной системой. Другой способ состоит в том, чтобы считать опции непосредственно из командной строки. Например:

```
grep pirntf *.c
```

Эта команда оболочки ищет орфографические ошибки в вашем исходном тексте на языке C. Команда имеет два параметра командной строки: `pirntf` и `*.c`. Эти две строки текста передаются главной функции `main()` как параметры, которые программа может вычислить и затем выполнить действия с ними так же, как с параметрами, передаваемыми любой другой функцией.

Чтобы уяснить, как информация передается главной функции `main()`, вы должны знать больше о массивах и указателях (надеюсь, вы прочитаете об этом позже).

Доступ к файлам на диске

Компьютер может хранить информацию и обрабатывать ее позже. В языке С предусмотрен широкий набор разнообразных функций, которые могут читать информацию с дисководов и записывать ее на дисковод. Используя язык С, вы можете хранить данные на диске так же, как это делает любая программа, которая имеет команду File⇒Save (Файл⇒Сохранить). Правда, в программе на С вам самим придется написать код команды File⇒Save (Файл⇒Сохранить).

Взаимодействие с операционной системой

Язык С также позволяет выполнять функции операционной системы. Вы можете изменять (заменять) каталоги, создавать новые каталоги, удалять файлы, переименовывать файлы и выполнять много других нужных задач.

Вы можете также из ваших программ выполнить другие программы — иногда две сразу! Кроме того, ваша программа может выполнить команды операционной системы и получить результаты их выполнения.

Наконец, вы можете заставить вашу программу взаимодействовать со средой и исследовать состояние вашего компьютера. Вы можете даже выполнять услуги (сервисы, службы) или бродить по сети. Почти все, что может сделать компьютер, вы можете добавить в вашу программу и сделать это непосредственно в программе.

Разработка больших программ

Не будет ничего неправильного, если файл исходного текста будет содержать 50 000 строк. Правда, потребуется больше времени для его компиляции, да и поиск ошибок займет определенное время, поэтому я бы не рекомендовал использовать такие большие файлы.

Лучший подход к записи больших программ состоит в том, чтобы разбить программу на меньшие модули. Например, один модуль может содержать дисковые подпрограммы; другой — инициализацию; а третий — подпрограммы, отображающие информацию на экране. Эта стратегия не только упрощает отладку, но и — это страшная тайна — если вы делаете модули достаточно хорошо, вы можете многократно использовать их в других программных проектах. Это определенно экономит время.

В языке С при компиляции каждого файла исходного текста генерируется файл объектного кода. Объектные файлы затем связываются с различными библиотеками, чтобы сгенерировать окончательный исполняемый файл. Эта технология работает весьма хорошо. Фактически, вы можете совместно использовать переменные в различных модулях исходного текста, вызывать функции в других модулях и делать много других полезных вещей.

Десять подсказок для подающего надежды программиста

В этой главе...

- Использование команды history (хронология)
- Держите ваш редактор открытым в другом окне
- Преимущества цветового кодирования в редакторе
- Используйте команды с номерами строк в вашем редакторе
- Держите окно приглашения к вводу команды открытым
- Еще несколько команд
- Обозначение переменных
- Решение загадок приращения и уменьшения
- Выход из цикла

В

этой главе я поместил некоторые из моих наилучших советов для начинающих программистов. К сожалению, я не имел возможности воспользоваться подобными советами, когда компьютеры были паровыми, ведь я начал осваивать программирование именно в те дни.

Команда history (хронология) позволяет использовать хронологию командной строки

Переключаться между вашим редактором, компилятором и приглашением к вводу команды весьма утомительно. К счастью, большинство оболочек командной строки (и в Unix, и в Windows) имеет средство повторения предыдущих команд. Используйте его!

Например, если вы нажимаете клавишу со стрелкой “вверх”, вы можете повторно вызвать предыдущую командную строку. Нажимая повторно клавишу со стрелкой “вверх”, вы снова вызовете команду, выполненную перед этим. Если вы много редактируете, используйте клавишу со стрелкой “вверх”, чтобы повторно вызвать команду редактирования, т.е. запустить ваш редактор и вызвать команду перетрансляции программы.

Держите ваш редактор открытым в другом окне

Предыдущий совет может быть не наилучшим (и даже может быть неприемлемым), если вы используете графическую среду вашей операционной системы. Держите приглашение к вводу команды открытым в одном окне, а ваш текстовый редактор — открытым в другом окне. Если вы не забываете сохранять исходный текст, можно открыть отдельное окно для редактора и редактировать в нем файл исходного текста. А в другом окне, пользуясь приглашением к вводу, можно выполнять команды компиляции исходного текста, а также выполнять получаемую программу.

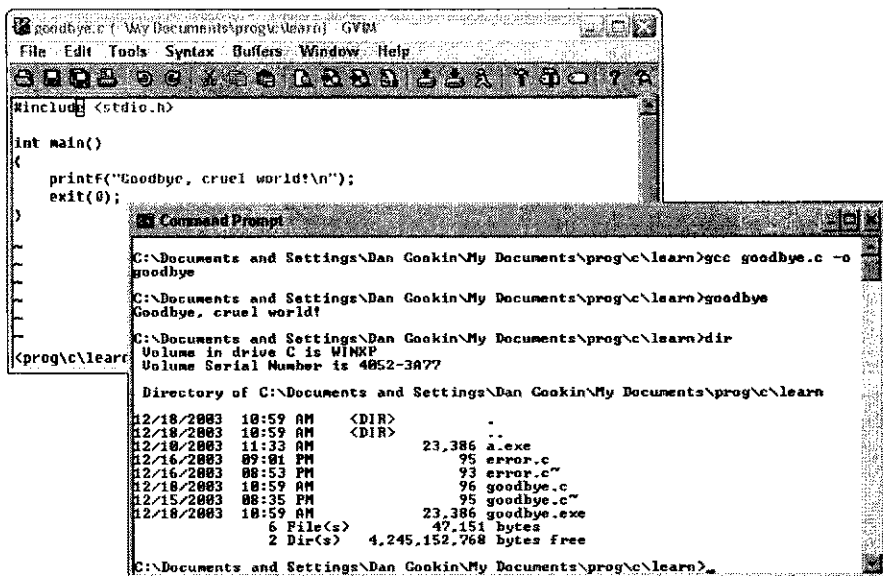


Рис. 28.1. Переключение между окном редактора и окном приглашения к вводу команды

Переключение между окнами выполняется щелчками мыши в том или ином окне. Кроме того, для переключения окон можно использовать клавишу <Alt+Tab>, которая работает в большинстве графических сред.



Не забывайте сохранять документ в одном окне перед компиляцией в другом окне.

Используйте текстовый редактор с цветной раскраской контекста

Позвольте объяснить, почему я рекомендую редактор Vim. Если имя файла исходного текста заканчивается на .c, Vim распознает код языка C и использует различные цвета для раскраски кода на экране. Эта особенность настолько полезна, что, если вы возвратитесь к одноцветному редактору, вы обратите внимание, что он замедляет вашу работу!



- ✓ Чтобы активизировать цвета в Vim, напечатайте двоеточие :, а затем **syntax enable** (включить синтаксис) и нажмите Enter.
- ✓ Когда вы выполняете Vim в окне Windows, выберите Syntax⇒Automatic (Синтаксис⇒Автоматически), чтобы были подсвечены ключевые слова языка C.
- ✓ В Unix, чтобы держать активизированной опцию **syntax enable** (включить синтаксис), отредактируйте или создайте в вашем основном каталоге файл, названный **.vimrc**. В этот файл добавьте следующую команду:
`:syntax enable`
(включить синтаксис)
- ✓ Затем снова сохраните файл **.vimrc** на диске.
- ✓ Еще одно преимущество подсветки текста состоит в том, что вы можете легко определить отсутствие кавычек. Дело в том, что текст между кавычками кодируется цветом, так что если кавычка отсутствует, исходный текст будет иметь необычный цвет (например, блеклый).
- ✓ Включите автоматические отступы вправо, если ваш редактор имеет такую особенность. Vim автоматически включает отступы вправо, когда вы используете допустимую синтаксисом команду или же из меню выбираете Syntax⇒Automatic (Синтаксис⇒Автоматически).

Используйте в вашем редакторе команды, в которых применяются номера строк

Компилятор с языка C в сообщениях об ошибках в вашем исходном тексте указывает строки, в которых были обнаружены ошибки. Если ваш текстовый редактор отображает номера строк, вы можете легко определить местонахождение строки, содержащей ошибку, и затем устранить ошибку.



- ✓ В Блокноте (Notepad) Windows вы можете отобразить номер строки и столбца в строке состояния, однако сначала убедитесь, что выключена опция Word Wrap (Перенос по словам). Для этого в случае необходимости достаточно выбрать Format⇒Word Wrap (Формат⇒ Перенос по словам), а затем выберите View⇒Status Bar (Вид⇒Строка состояния). (Обратите внимание, что команда Status Bar (Строка состояния) может быть недоступна в более ранних версиях Блокнота (Notepad).)
- ✓ Vim отображает позицию курсора справа внизу окна. (За номером строки следует запятая и номер столбца. Например, позиция курсора на рис. А.1 в приложении А “Прежде чем вы приступите к программированию” отображается как 1, 1.)
- ✓ В Vim для перехода к определенной строке применяется команда **G** (go to — идти). Например, если компилятор сообщает об ошибке в строке 64, напечатайте **64G**, и VIM немедленно перейдет к строке 64. Думайте про себя так: “Номер строки, Goto”, чтобы запомнить эту команду.

Держите открытым окно приглашения к вводу команды, если используете интегрированную среду разработки

В Windows большинство компиляторов поставляется с интегрированной средой разработки (Integrated Development Environment (IDE) — интегрированная среда разработки). Это — окно, которое содержит редактор, а также другие инструменты для того, чтобы создавать программы в Windows. Интегрированная среда разработки может быть весьма сложной, но пользоваться ею очень удобно; например, подобно Vim, интегрированная среда разработки Dev-C++ применяет цветовые коды для команд C.

Следует отметить, что большинство интегрированных сред разработки, как правило, не отображают вывод пультовых программ, а именно такие программы создаются в этой книге. По этой причине в случае использования интегрированной среды разработки удобно держать открытым окно, отображающее пульт, и переключаться на папку `prog\c\learn` в Windows. Тогда вы сможете печатать названия (имена) создаваемых вами программ и видеть их вывод в этом окне. В противном случае вывод не будет видим из интегрированной среды разработки.

Несколько удобных команд в окне с приглашением к вводу команды

Я настоятельно рекомендую ознакомиться с использованием приглашения к вводу команды. Хотя существуют сотни команд, которые можно ввести в приглашении к вводу команды, и еще миллиарды их вариантов и еще большее количество способов сделать опечатки (особенно если печатать в перчатках сварщика), все же обратите внимание на несколько команд, перечисленных в табл. 28.1.

Таблица 28.1. Команды в приглашении к вводу команды

Windows/DOS	Unix	Назначение команды
<code>dir</code>	<code>ls</code>	Перечисляет все файлы в текущем каталоге или папке. В Unix переключатель <code>-l</code> после <code>ls</code> используется для отображения подробностей о файлах в каталоге
<code>dir *.c</code>	<code>ls *.c</code>	Перечисляет только файлы исходных текстов на языке C в папке или в каталоге
<code>cls</code>	<code>clear</code>	Очищает экран
<code>ren</code>	<code>mv</code>	Переименовывает файл; после <code>ren</code> или <code>mv</code> следует первоначальное название (имя) файла, а затем — новое название (имя). Вот пример: <code>ren goodbye.c bye.c</code> или <code>mv goodbye.c bye.c</code> — эти команды переименовывают файл <code>goodbye.c</code> на <code>bye.c</code>
<code>cd</code>	<code>pwd</code>	Отображает название (имя) текущего каталога или папки. Используйте эту команду, чтобы убедиться, что вы находитесь в каталоге <code>prog/c/learn</code>
<code>type</code>	<code>cat</code>	Отображает содержимое текстового файла на экране; после <code>type</code> или <code>cat</code> следует название (имя) файла, который вы хотите отобразить. Вот примеры: <code>type source.c</code> и <code>cat source.c</code> — печать файла <code>source.c</code>

Windows/DOS	Unix	Назначение команды
del	rm	Удаляет файл; за del или rm следует название (имя) удаляемого файла. Вот примеры: команды del bye.c и rm bye.c удаляют файл bye.c
exit	exit	Выход — закрывает окно приглашения к вводу команды и закрывает терминал

Обратитесь к справочной литературе, где описывается приглашение к вводу команды, чтобы подробно изучить эти и другие команды, которыми удобно пользоваться в процессе создания программ.

Правильно называйте ваши переменные

Хотя в этой книге используется много односимвольных имен переменных, отнеситесь со всей ответственностью к выбору обозначений переменных в ваших собственных программах. Например, `x` — хорошее имя для переменной, но `counter` (счетчик) — намного лучше.

Такое замечание может показаться глупым для крошечных программ, но в больших программах бывает очень трудно убедиться, что `x` или иная объявляемая вами переменная не используется в некоторой другой части программы. Плохая новость!

Загадки пост- и предприращения и уменьшения

Конечно, операторы `++` и `--` удобны для приращения и уменьшения значения переменной. Однако не забывайте о том, что все инструкции языка C нужно записывать в одной строке, а также о том, что `++` и `--` перед именем переменной выполняют свою задачу до использования значения переменной. Если вы помещаете `++` или `--` после имени переменной, операция будет выполнена позже.

Обратитесь к главе 25 “Математическое безумие!”, чтобы быстро вспомнить все, что относится к данной теме.

Выход из цикла

Все циклы должны иметь точку выхода. Независимо от того, планировали вы определить эту точку в инструкции управления циклом или установить в цикле с помощью команды `break`, убедитесь, что она там есть!

Я вспоминаю много случаев, когда я сидел за компьютером и ждал, когда же программа “пробудится”, только затем, чтобы потом понять, что она застряла в цикле, который я запрограммировал без точки выхода. Особенно просто просмотреть этот недостаток в большой программе с “длинными” циклами; если высота исходного текста цикла `for` больше, чем высота окна текстового редактора, очень просто потерять нить рассуждений.

Десять способов самостоятельно разрешить свои проблемы в программах

В этой главе...

- Исправляйте только одну ошибку за один раз
- Разделяйте код
- Упрощайте свою задачу
- Обсуждайте свои проблемы
- Устанавливайте контрольные точки
- Следите за переменными
- Не забывайте о документации
- Используйте средства отладки
- Используйте оптимизатор
- Читайте книги!

Добро пожаловать в мир отладки. За всю свою жизнь я встретил только одного программиста, который смог сесть и закодировать весь проект на пустом месте и сделать это так, чтобы он компилировался и работал с первого раза. Как оказалось, этот программист был способен сделать это только однажды — и это был тот проект, с которым он был хорошо знаком. Хотя он — один из лучших программистов во всем мире, мечта о записи, компиляции и выполнении всего проекта именно в таком порядке остается мечтой для большинства программистов.

Да, программы полны ошибок. Встречаются опечатки, которые замечает компилятор, но в программах также есть и другие ошибки (или дефекты). При компиляции и редактировании связей ошибки в программе не обнаруживаются, но когда программы работают, происходят совершенно неожиданные вещи. Да, все программы выполняют распоряжения программиста. Часто случается, что программист забывает о важных вещах. И тогда вы должны обратиться к этой главе и прочитать обзор моих десяти способов решить ваши собственные проблемы в программах. Сделайте это до того, как будете звонить, посылать письмо по электронной почте или выносить вашу проблему на публичное обсуждение. Публика будет благодарна вам!

Исправляйте только одну ошибку за один раз

Исправляйте ошибки (дефекты) по одной. Даже если вы знаете, что программа имеет несколько неправильных инструкций, установите систему их исправления.

Например, вы обращаете внимание, что заголовок расположен слишком далеко вправо, внизу экрана появляются случайные символы, а методы прокрутки не перемещают наивысшую строку. Избегайте искушения решить все три проблемы за один сеанс редактирования. Вместо этого разрешите одну проблему. Скомпилируйте программу и запустите ее на выполнение, чтобы увидеть, как она работает. Затем разрешите следующую проблему.

Проблема, с которой вы сталкиваетесь, когда пробуете решить сразу слишком много задач, состоит в том, что вы можете ввести новые ошибки. Однако удерживать все под контролем будет проще, если вы помните, что изменяли, например, только строки 173 и 174 вашего исходного текста.

Разбивайте код на части

Если исходный текст становится длинным, постарайтесь разбить его на части, точнее на отдельные модули. Я не рассматриваю эту тему в данной книге (вероятно, это не та проблема, с которой вы встретитесь в ближайшем будущем), тем не менее, разбиение на модули может действительно упростить отслеживание и исправление ошибок (дефектов).

Даже если вы не используете модули, используйте комментарии, чтобы помочь визуально разбить код на отдельные разделы. Попробуйте объявить цель каждого раздела. Например¹:

```
/*
Verification function
-----
This function takes the filename passed to it
and confirms that it's a valid filename and
that a file with that name doesn't already exist.
Returns TRUE/FALSE as defined in the header.
*/
```

```
/*
Функция проверки
-----
Эта функция принимает переданное ей имя файла filename и проверяет,
что это правильное (допустимое) имя файла и что файл с этим названием
(именем) еще не существует.
Возвращает TRUE (ИСТИНА) или FALSE (ЛОЖЬ), как определено в
заголовочном файле.
*/
```

Я также помещаю разрыв между функциями просто для того, чтобы визуально их разделить:

```
/*
```

Упрощение

Избегайте искушения записать несколько команд в одной строке, если вы не уверены, что данная часть кода работает хорошо. Есть много причин, по которым следует придерживаться данного совета, но наиболее важной из них является та, что это облегчает чтение кода. Чем легче читается ваш код, тем проще другим программистам (да и вам) найти ошибку.

¹ Если программы будут обрабатываться системами, в которых символы кириллицы могут быть искажены, комментарии лучше писать по-английски. Такое же требование предъявляется часто и иностранными заказчиками. — Прим. ред.

Например, вместо такой строки:

```
while(toupper(getch())!=ZED)
```

лучше написать так:

```
while(ch != ZED)
{
    ch = getch();
    ch = toupper(ch);
    /* и так далее */
}
```

Да, может потребоваться еще несколько строк, но если вы получите сообщение об ошибке в одной из этих строк, вы точно знаете, куда надо смотреть. А строка `while(toupper(getch())!=ZED)`, возможно, вызовет сомнения в нескольких потенциально опасных конструкциях.

Кроме того, второй пример легче читается другими программистами.



Наконец, когда будете уверены, что ваш код работает, можете уплотнить его и записать все на одной строке — если вы захотите, конечно. Помните, что цель состоит в том, чтобы заставить программу работать, а не пробовать произвести впечатление на других программистов с помощью непонятного исходного текста.

Обсуждение программы

Один из лучших способов мгновенно находить ошибки (дефекты) состоит в том, чтобы показать ваш код другому программисту. Но ведь другой программист не знаком с тем, что вы делаете. Вы начинаете рассказывать, что вы делаете — и тем самым обсуждаете логику вашего кода. Вы объясняете, что делает та или иная часть программы и почему вы сделали тот выбор, который вы сделали.

Внезапно, когда вы объясняете, что вы сделали, вы обнаруживаете, что некая часть вашего кода не соответствует вашим намерениям. Вы говорите “Ага!”. Другой программист кивает в ответ, и вы можете продолжать вашу работу.

Эта методика обсуждения программы работает независимо от присутствия другого программиста.

Устанавливайте контрольные точки

Предположим, вы знаете, что ошибка (дефект) находится в функции `windshield()`, но не знаете, где именно. Ошибка (дефект) скрывается в начале кода? В подпрограммах инициализации? Как раз перед многочисленными вызовами математических функций? Где-то в конце? Где? Где? Где?

Один из способов узнать это состоит в том, чтобы поместить в программу контрольные точки. Этот способ позволяет отладить даже за цикливающуюся программу. Вставьте в некоторой точке функцию `return()` или `exit()`, которая немедленно останавливает программу. Этим способом вы сможете как бы уменьшить ваш противный ужасный код. Если программа остановится в контрольной точке, ошибка находится в программе дальше. Если программа не останавливается в контрольной точке, ошибка находится до контрольной точки.

Контролируйте ваши переменные

Иногда программа выполняется так, вроде она сошла с ума, потому что значения, которые вы ожидали найти в переменных, находятся где угодно, только не там. Чтобы убедиться, что в переменных нет никаких возмутительных непредусмотренных значений, вставьте инструкцию `printf()`, чтобы отобразить значения переменных на экране. Не расстраивайтесь, если эта методика испортит вывод на дисплей; цель — отладка.

Например, однажды я столкнулся с программой, которая постоянно за цикливалась. Я не мог выяснить, почему она бесконечно повторялась. Обсуждение исходного текста ничего не дало, но после того как я вставил инструкцию `printf()`, которая отображала значение переменной в ходе выполнения цикла, я заметил, что это значение весело перескакивало через значение, при котором должно было выполниться условие окончания цикла; это значение продолжало увеличиваться до бесконечности и потому условие окончания цикла никогда не выполнялось. Я добавил простой условный оператор, чтобы устранить проблему, и программа выполнялась просто великолепно.

Документируйте вашу работу

В университете о комментариях говорят еще чаще, чем о жвачке в театре. Комментируйте это, комментируйте то! Я никогда не забуду проекты однокурсников длиной в три страницы, причем половину проекта занимали глупые комментарии в начале программы. Такая ерунда никого не впечатляет.

И в самом деле, документируйте вашу работу, только учтите, что документация состоит из примечаний к будущей версии. Это — напоминание вроде “вот что я думал” или “вот как я это планировал”. Совсем не обязательно документировать каждую небольшую глупую часть программы. Следующий комментарий бесполезен:

```
i++; /* добавить 1 к значению i */
```

Вот гораздо лучший комментарий:

```
/* Поскольку нуль - первый элемент, здесь увеличиваю переменную i, чтобы не смущать пользователя. */  
i++;
```

Комментарии могут помочь также создать будущую версию программы:

```
/* Это - подпрограмма начисления выплат. В будущем здесь было бы неплохо добавить звуковой эффект, скажем, звон монет в кармане или звук автомата, выдающего деньги. */
```

Комментарии могут также быть примечаниями, указывающими на то, что нужно исправить. Они могут напомнить о том, что прежде шло не так, как надо, и давать подсказки насчет того, как разрешить проблему:

```
/* я не могу вставить здесь нулевой адрес. Предполагается, что подпрограмма работает, но она упорно продолжает возвращать нулевое (пустое) значение. Приложение С книги Дэвиса содержит первоначальную подпрограмму. */
```

Используйте средства отладки

Если ваш компилятор имеет отладчик, используйте его! Отладчики позволяют рассматривать, как выполняется программа, контролировать при этом переменные и следить за тем, как компьютер пошагово выполняет код.

Проблема с отладчиками состоит в том, что вы обычно должны компилировать специальную версию программы, чтобы она содержала отладочный код. Это делает программу такой огромной и такой вялой! Если вы используете отладчик, не забудьте исключить отладочный код при компиляции окончательной версии программы.

Используйте оптимизатор кода на C

Многие весьма забавные инструментальные средства оптимизатора исследуют ваш код и делают предложения по оптимизации. Чаще других обычно применяется один весьма общий инструмент — `lint` (читается *линт*). Существуют также и другие инструментальные средства; на любом Web-сайте, посвященном программированию на C, есть целый список разнообразных средств такого типа, притом многие из них — с открытым кодом (*open source*). (Спасибо сообществу разработчиков программ с открытым кодом — *open source community*).



- ✓ Точно так же, как программа проверки орфографии не делает вас лучшим писателем, оптимизаторы не делают вас лучшим программистом. Однако они могут помочь улучшить ваш код так, чтобы он выполнялся быстрее.
- ✓ Доступны также и другие инструментальные средства программирования. Фактически, вся операционная система Unix разработана для программирования. Если у вас есть время, изучите следующие инструментальные средства: `touch`, `make` и `grep`.

Читайте книги!

Программисты-новички часто спрашивают меня по электронной почте: “Какую литературу вы бы посоветовали читать после вашей книги?” Очевидно, что лучше всего выбрать сопровождающую книгу *C All-in-One Desk Reference For Dummies*, выпущенную издательством “Wiley”. Но реальный ответ такой: “Это зависит от того, что вы собираетесь программировать”. Если вы хотите программировать игры, изучайте книги по программированию игр. Вы можете читать книги по программированию сетей, программированию операционных систем, программированию графики и по многим другим разделам программирования. Некоторые книги, вероятно, придется поискать в университетских книжных магазинах, но главное — помнить, что они существуют.

- ✓ После прочтения этой книги вы будете знать приблизительно 95 процентов языка программирования C++. (Легче будет освоить и другие языки программирования.) После изучения этой книги было бы естественным заняться языком программирования C++. Но рассмотрите также возможность изучения таких языков, как Perl (читается *Перл*), Java (читается *Ява*), Python (читается *Питон*) и других.

- ✓ Если вы собираетесь заняться Web-программированием, советую вам также изучить язык создания Web-страниц PHP.
- ✓ И главное — пишите программы. Практикуйтесь. Разрабатывайте новые проекты и воплощайте новые идеи. Программирование, по мнению многих людей, подобно решению увлекательных задач. Решайте задачи. Развлекайтесь!

Прежде чем вы приступите к программированию

В этом приложении...

- Конфигурирование компьютера в качестве рабочей станции программиста
- Выбор компилятора
- Выбор редактора
- Создание программ

Прежде чем можно будет программировать на языке C на вашем компьютере, нужно выполнить несколько условий. Цель данного приложения состоит в том, чтобы указать, какие инструменты нужны и как их использовать для работы с программами, приведенными в этой книге. Это совсем несложно, но может показаться непривычным, так что будьте внимательны!

Настройка и установка

Для программирования на C на вашем компьютере должны быть:

- ✓ компилятор;
- ✓ место, куда вы будете помещать свои программы.

В операционные системы Linux, Unix и Mac OS X компилятор с языка C уже включен; он поставляется с операционной системой. Для Windows и старых компьютеров Mac вы должны получить компилятор. Это не столь трудно, как кажется.

Компилятор с языка C

Язык C весьма популярен, поэтому очень много компиляторов подойдут для программ из этой книги. Однако я рекомендую следующие.

Windows. Если вы работаете на компьютере с Windows, я рекомендую использовать компилятор с языка C, совместимый с GCC. Список компиляторов приведен на Web-странице этой книги www.c-for-dummies.com.

Для программ из этой книги я использовал компилятор MinGW, который поставляется с интегрированной средой разработки Dev-C++¹. Это бесплатный компилятор и доступ к нему открыт на странице www.bloodshed.net.

Какой бы компилятор вы ни использовали, обратите внимание на его местоположение на жестком диске вашего ПК. Укажите это местоположение при создании пакетного файла или измените системный путь так, чтобы вы могли обратиться к компилятору из любой папки на вашей дисковой системе. Подробнее об этом позже.

¹ Интегрированная среда разработки (Integrated Development Environment, IDE) — это комплекс инструментальных средств, предназначенных для разработки программ. — Прим. ред.



- ✓ Есть и другие компиляторы — например, пользующийся большим спросом Microsoft Visual C++ (MSVC). Компилятор MSVC вполне подходит для выполнения программ из этой книги. Я не знаком с текущей версией MSVC и не ссылаюсь на него в этой книге, в силу этих обстоятельств я не могу ответить на вопросы о нем по электронной почте. Если у вас нет MSVC, я не вижу никаких разумных причин покупать его.
- ✓ В Internet доступно множество бесплатных, условно-бесплатных компиляторов с языка C, а также компиляторов с открытым кодом (open-source).
- ✓ Если у вас есть другие книги по языку C, возможно, в одной из них имеется компакт-диск с бесплатным компилятором.
- ✓ Лучше всего для этой книги подойдет любой компилятор с языка C, совместимый с GCC или с GNU.

Linux, FreeBSD или Mac OS X. Если вы используете любую из этих разновидностей Unix, вы должны заранее установить компилятор GCC и подготовить его к использованию. Чтобы убедиться в этом, откройте окно терминала и напечатайте следующую строку в приглашении к вводу команды:

```
gcc -v
```

На экране отобразится номер версии GCC и другая информация. Если вы получите ошибку `Command not found` (Команда не найдена), GCC не установлен; вы должны изменить вашу операционную систему — установить (включить в нее) GCC, а также все библиотеки программ на C и другие материалы. (Все это вы можете сделать с помощью программы установки, имеющейся в вашей операционной системе, или с помощью программы конфигурации: обычно для этого не требуется полная повторная установка операционной системы.)

Unix. Если у вас “настоящая” версия Unix, нужная вам команда — `cc`, а не `gcc`. На самом деле `cc` работает и на других операционных системах семейства Unix, где команда `cc` часто вызывает компилятор GCC, что сделано для совместимости.

Mac (до OS X). В старых версиях Mac нет встроенного компилятора с языка C. Я рекомендую компилятор Code Warrior (Воин Кода), но лучше всего посетить Web-узел компании Apple <http://developer.apple.com/>, чтобы увидеть, доступны ли какие-нибудь другие (бесплатные) компиляторы.

Место для хранения проектов

Когда вы учитесь программировать, вы создаете огромное количество файлов. Эти файлы включают первоначальные текстовые файлы исходного текста, окончательные программные файлы, и, возможно, даже файлы объектного кода — все зависит от компилятора. Очевидно, необходимо организовать хранение этих файлов отдельно от вашего обычно используемого барахла.

Для этой книги я рекомендую создать папку или каталог `prog` (программы). Создайте эту папку в вашей основной папке — папке `$HOME` в Unix или в папке My Documents (Мои документы) в Windows. Папка `prog` предназначена для хранения всех ваших программных проектов.

В папке `prog` создайте папку `c` для всех программных проектов на языке C.

Наконец, создайте папку `learn` (учебная), в которую вы будете помещать все проекты из этой книги. В остальной части этого приложения приведены примеры.

Windows. Чтобы создать папку для проектов на языке C, выполните следующие шаги.

1. Откройте папку **My Documents** (Мои документы) — для этого достаточно щелкнуть на пиктограмме, имеющейся на рабочем столе.
2. Выберите **File⇒New⇒Folder** (Файл⇒Создать⇒Папку), чтобы создать новую папку, и затем назовите папку **prog**.
3. Откройте папку **prog**.
4. Выберите **File⇒New⇒Folder** (Файл⇒Создать⇒Папку), чтобы создать новую папку, и затем назовите папку **c**.
5. Откройте папку **c**.
6. Создайте папку в папке **c** и назовите созданную папку **learn** (учебная).
7. Закройте окно папки **c**.

В папке **learn** вы будете размещать все файлы, создаваемые в примерах из этой книги. **Linux, FreeBSD, Mac OS X** или **Unix**. Чтобы создать папку для проектов на языке **C**, выполните следующие шаги.

1. Если вы используете графическую оболочку, откройте окно терминала. Вы должны добраться до приглашения к вводу команды.

Окно терминала должно открыться в основном каталоге вашей учетной записи. Если вы не находитесь в вашем основном каталоге, напечатайте команду **cd**, чтобы возвратиться туда.

Убедитесь, что при входе вы не использовали учетную запись **root**, поскольку при создании программ пользователем **root** повышается риск нарушения защиты.

2. Создайте ветвь каталогов **prog/c/learn**:

```
mkdir -p prog/c/learn
```

Переключатель **-p** указывает, что **mkdir** должна создать все подкаталоги, которые определены в команде; это то же самое, что выполнение сразу трех отдельных команд **mkdir**. Одной командой вы создали каталог **prog**, подкаталог **c** и, наконец, подкаталог **learn**.

Вы используете папку **learn** для хранения всех исходных текстов и программных файлов, созданных в примерах из этой книги.

Mac (до OS X). Увы, в старых операционных системах Mac не была предусмотрена “домашняя папка” для хранения всех ваших материалов. Если у вас есть такая папка, используйте ее в качестве корневой, в ней вы будете создавать подпапки в следующих шагах. В противном случае вы можете создавать эти папки прямо на рабочем столе для удобства доступа.

1. Нажмите **⌘+N**, чтобы создать новую папку.
2. Назовите созданную папку **prog**, она предназначена для хранения программ.
3. Откройте папку **prog**.
4. Нажмите **⌘+N**, чтобы создать подпапку в папке **prog**.
5. Назовите эту папку **c**.
6. Откройте папку **c**.
7. Нажмите **⌘+N**, чтобы создать подпапку в папке **c**.



8. Назовите эту подпапку `learn`.

9. Закройте все открытые окна, которые только что были созданы.



Компилируя программы, не забывайте сохранять все ваши файлы в папке `learn`.

Создание программы

Чтобы получать программы, нужны два инструментальных средства: редактор и компилятор. Редактор позволяет создать и редактировать исходный текст, который является просто текстовым файлом. Затем компилятор волшебным образом преобразовывает этот текст (переводит его на язык, который понимает компьютер) и записывает результат преобразования в файл программы.

Методы программирования в этой книге иллюстрируются на специально подобранных примерах маленьких программ, которые содержат типичные примеры инструкций на языке C. По этой причине для компиляции программы иногда легче использовать приглашение к вводу команды, чем интегрированную среду разработки, которая, возможно, была поставлена вместе с вашим компилятором. Я рекомендую познакомиться с применением приглашения к вводу команды.

Следующие шаги не применимы при программировании на ПК Macintosh с операционной системой более старой, чем OS X. Если вы используете старый Mac, обратитесь к документации по вашему компилятору, чтобы узнать, как редактировать и компилировать программы. Не забывайте сохранять все создаваемые вами файлы в специально созданной вами папке `learn`.

Поиск каталога (папки) `learn`

Первый шаг к программированию состоит в том, чтобы научиться находить каталог (или папку) `learn`, используя приглашение к вводу команды. Выполните следующие шаги:

1. Запустите окно с приглашением к вводу команды или терминал.

В Windows выполните программу `CMD.EXE`, называемую также подсказкой MS DOS.

Эта программа находится в Accessories (Вспомогательные программы) или в основном меню Programs, открываемом с помощью кнопки Start (Пуск). Кроме того, вы можете напечатать `CMD` в диалоговом окне Run (Выполнить, Запуск программы), чтобы запустить окно приглашения к вводу команды.

В Linux, OS X, FreeBSD и других операционных системах семейства Unix откройте терминал, если вы используете графическую оболочку. В противном случае можете использовать любой терминал.

2. Перейдите в ваш основной каталог.

В Windows XP напечатайте эту команду:

```
cd "my documents"
```

```
(cd "Мои документы")
```

В других версиях Windows напечатайте эту команду:

```
cd "%My Documents"
```

```
(cd "%Мои документы")
```

Приглашение к вводу команды должно теперь отразить, что вы используете папку My Documents (Мои документы). На терминале появится что-нибудь вроде:

```
C:\Documents and Settings\Dan\My Documents>  
C:\Documents and Settings\Dan\Мои документы>
```

или:

```
C:\My Documents>  
C:\Мои документы>
```

(В последней части подсказки будет “My Documents” или “Мои документы”).

В Linux, FreeBSD или Mac OS X напечатайте команду `cd`, чтобы перейти в ваш основной каталог. Все это делает одна команда.

3. Перейдите в каталог **learn**.

Для этого напечатайте:

```
cd prog/c/learn
```

Правда, в старых версиях Windows вместо этого надо напечатать

```
cd prog\c\learn
```

(Обратите внимание, что пришлось использовать наклонные черты влево, а не косые черты.)

4. Убедитесь, что вы находитесь в надлежащем каталоге.

Чтобы сделать это в Windows, напечатайте команду `cd`; в Unix напечатайте `pwd`. Будет отображен текущий каталог, который должен напоминать какую-нибудь из следующих строк:

```
C:\Documents and Settings\имя\My Documents\prog\c\learn  
C:\Documents and Settings\имя\Мои документы\prog\c\learn  
C:\My Documents\prog\c\learn  
C:\Мои документы\prog\c\learn  
/home/пользователь/prog/c/learn  
/Users/пользователь/prog/c/learn
```

Обратите внимание, что общая часть этих строк — это их последняя часть, `prog/c/learn`. Если вы увидите ее (или `prog\c\learn`), можете начать работу.

Каталог `learn` — та папка, в которой вы сохраняете все, что описано в этой книге. Именно в этом каталоге размещаются файлы, которые вы редактируете, создаете, компилируете и над которыми вы выполняете другие операции.

Запуск редактора

Чтобы создать исходный текст на языке C, вы должны использовать текстовый редактор. В Windows вы можете использовать команду `EDIT`, чтобы вызвать редактор MS DOS `edit`. Он довольно прост, и использовать его совсем несложно, он работает с мышью, поставляется бесплатно и доступен в любую минуту.

В различных операционных системах Unix есть несколько редакторов. Самый простой текстовый редактор — Easy Editor (Простой Редактор), активизируемый командой `ee`. Впрочем, вы можете использовать любой редактор системы Unix — их имеется довольно много.

Мой любимый редактор для редактирования программ на C — `vim`, это вариант бесславного редактора `vi` системы Unix (см. рис. А.1). В отличие от `vi`, `vim` использует цвета для кодирования текста. Когда вы редактируете исходный текст с помощью `vim`, ключевые слова, значения и другие части языка C подсвечиваются определенными цветами.

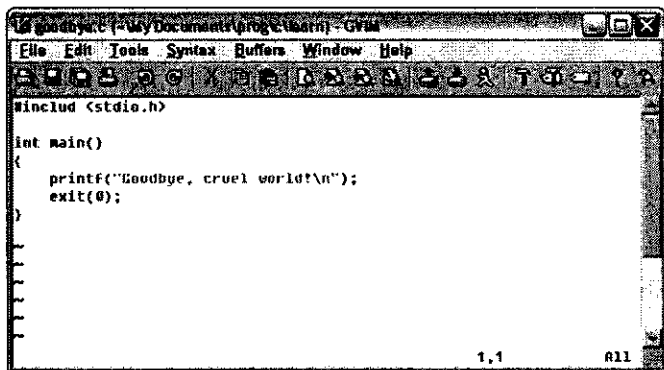


Рис. А.1. Редактор vim



- ✓ Версии vim существуют для Linux, FreeBSD, Mac OS X, Windows и даже для старых компьютеров Mac. Вы можете найти их по адресу www.vim.org.
- ✓ Windows XP может не любить команду EDIT. В качестве альтернативы редактировать исходный текст можно с помощью Notepad (Блокнот). Например, чтобы редактировать текстовый файл GOODBYE.C, напечатайте следующую команду после подсказки:
NOTEPAD GOODBYE.C
(Блокнот GOODBYE.C)
Notepad (Блокнот) откроется в другом окне, где вы можете редактировать текстовый файл. Просто закройте окно, когда все закончите.

Компиляция и редактирование связей

После создания текстового файла — исходного текста, его необходимо скомпилировать и отредактировать связи. На этом этапе обычный текстовый файл преобразовывается в исполнимую программу, которую может выполнить компьютер.

Прочитайте надлежащий подраздел документации, где описывается, как компилировать и редактировать связи в вашей операционной системе. Для старых компьютеров Mac, на которых еще не было OS X, все это должно быть описано в справочниках, которые поставлялись вместе с вашим компилятором.

Запуск GCC в Windows

Несмотря на все усовершенствования, сделанные в Windows, для компиляции программ можно использовать и приглашение к вводу команды DOS. Так или иначе, результат будет тем же самым...

Компиляторы Windows не рассчитаны на компиляцию программ из командной строки. По этой причине нужно сделать так, чтобы компилятор можно было вызвать из любого приглашения к вводу команды и из каждой папки вашей компьютерной системы. Проще всего сделать это так. Нужно создать пакетный файл, который выполняет команду GCC (или другую), вызывающую компилятор. Это не совсем просто, но, к счастью, это должно быть сделано только однажды.

Далее предполагается, что вы уже установили среду Dev-C++ на вашем ПК. Кроме того, предполагается, что вы установили Dev-C++ в папку C:\Dev-C++.

(Если вы устанавливали Dev-C++ в другую папку, вы должны записать путь к этой папке. Например, если вы устанавливали Dev-C++ в папку Program Files (Программы), то путь будет таким: C:\Program Files\Dev-C++. Вы должны помнить путь!)

Глубоко вздохните.



1. Откройте окно MS DOS или другое окно с приглашением к вводу команды.

Вы не знаете как? Создайте ярлык окна MS DOS на рабочем столе — это сэкономит уйму времени, когда вы будете читать эту книгу. Подробности можете прочитать в вашей любимой книге по Windows.

2. Перейдите в папку Windows:

```
cd \windows
```

(Я предполагаю, что Windows — название (имя) вашей папки с системой Windows. Если это не так, замените название (имя) папки *windows* в команде `cd \windows` настоящим именем папки, например, WINNT или другим.)

В папке Windows создайте пакетный файл (это своеобразная программа) — ярлык команды GCC, которая нужна для Dev-C++. После этого вы сможете использовать команду GCC в любом приглашении к вводу команды в Windows.

3. Внимательно напечатайте `copy con gcc.bat` (копировать `gcc.bat`) и нажмите клавишу <Enter>.

4. Внимательно напечатайте следующую строку:

```
@c:\Dev-C++\bin\gcc %1 %2 %3 %4 %5 %6
```

Строка начинается со знака @. После него следует полное имя пути к компилятору GCC среды Dev-C++: `c:\Dev-C++\bin\gcc`. (Если вы установили GCC в другую папку, поместите ее путь в данную команду вместо `c:\Dev-C++\bin\gcc`; не забудьте заключить путь в двойные кавычки, если он содержит пробелы!)

После `gcc` следует пробел, а затем `%1`, пробел, `%2`, пробел, и так далее. Это важно!

Если вы сделаете ошибку, используйте клавишу <Backspace> (возврат на позицию), чтобы исправить ошибку.



Если все это кажется вам странным, попросите кого-нибудь, кто лучше понимает Windows (или DOS), помочь вам.

5. Проверьте строку.

Перепроверьте все. Только когда убедитесь, что все набрано как в книге, выполните следующий шаг.

6. Нажмите клавишу <Enter>.

7. Нажмите клавишу <F6>.

На экране появится ^Z.

8. Нажмите клавишу <Enter>.

Вы увидите `1 file(s) copied` (1 файл скопирован). Кроме того, был создан файл `GCC.BAT`.

Теперь вы должны проверить файл `GCC.BAT`, чтобы убедиться, что он работает. Выполните шаги, приведенные в этом приложении ранее, чтобы перейти в папку `learn`. (Может быть, проще закрыть текущее окно с подсказкой (Command Prompt) и открыть новое). Когда вы перейдете в папку `learn`, напечатайте следующую команду после подсказки:

```
gcc -v
```

Если вы увидите, что на экране появляется компилятор, поздравляю вас! Вы заставили его работать!

Если компилятор не работает, обратите внимание на предыдущие шаги. Вы, вероятно, не скопировали текст должным образом на шаге 4 или создали файл в другой папке. Повторите шаги и нажмите Y после шага 3, чтобы переписать командный файл GCC.BAT, т.е. заменить его новым.

Windows: компиляция, редактирование связей и запуск

После настройки файла GCC.BAT вы можете создавать программы. В конечном счете, вы повторяете следующие шаги достаточно часто, так что едва ли вам потребуется обращаться к этому приложению за справкой.

1. Убедитесь, что вы находитесь в надлежащей папке.

Если нужно, обратитесь к разделу “Поиск каталога (папки) learn” этого приложения.

2. С помощью текстового редактора создайте файл исходного текста. В главе 1 “Основы языка C” приведена распечатка программы GOODBYE.C. Напечатайте этот текст с помощью редактора в соответствии с инструкциями, приведенными в главе 1 “Основы языка C”.

3. Скомпилируйте исходный текст и отредактируйте связи.

Этот шаг выполняет команда GCC — обе операции сразу. Вот какую команду нужно напечатать:

```
gcc goodbye.c -o goodbye
```

Вы напечатали следующее:

- команду gcc, которая компилирует исходный текст и редактирует связи;
- название (имя) файла исходного текста goodbye.c;
- переключатель вывода -o;
- название (имя) полученной программы goodbye.

Если вы опустите переключатель -o и его опцию, в Windows GCC создаст программный файл по имени A.EXE. Я не рекомендую это. Помните об опции -o и укажите название (имя) выходной программы. Название (имя) может быть то же самое, что и у файла исходного текста. (Файлы исходного текста языка C заканчиваются на .C, а файлы программ — на .EXE.)

4. Выполните полученную программу.

Напечатайте название (имя) программного файла после подсказки. Для этого примера напечатайте goodbye (до свидания) и нажмите <Enter>. На этом шаге выполняется код созданной программы; она отображает сообщения на экране или делает что-нибудь более интересное — все зависит от того, как была разработана программа.

Это основные шаги, которые нужно выполнить (все в папке learn), чтобы создать программы, приведенные в этой книге. В конечном счете вы будете выполнять их автоматически.

- ✓ Если появятся сообщения об ошибках, вроде того, что компилятор не может найти свои файлы, или сообщения о некоторых проблемах при установке, обратитесь к документации по компилятору или на Web-узле за справкой. Я не могу помочь решить проблемы, связанные с компилятором или его установкой.



- ✓ Для каждой создаваемой программы есть два файла: файл исходного текста (текстовый файл) и программный файл.
- ✓ Некоторые компиляторы с языка C требуют выполнения двух шагов: сначала нужно программу скомпилировать, а затем (в отдельном шаге) скомпоновать. Для дополнительного шага на диске создается третий файл — OBJ, или объектный файл. В GCC компиляция объединена с компоновкой. После успешной компоновки GCC удаляет объектный файл.
- ✓ Я рекомендую сохранять программы для возможности получения справки в будущем; поэтому не удаляйте их, пока вы не освоите программирование хорошо и действительно больше не будете нуждаться в них.

Linux, FreeBSD и Mac OS X ОС: компиляция, редактирование связей и выполнение

В этой книге разработка примеров программ выполняется за три шага. В конечном счете вы повторяете эти шаги достаточно часто, так что вам не придется возвращаться, чтобы узнать, какой шаг будет следующим:

1. Убедитесь, что находитесь в папке `learn`.

Если нужно, обратитесь к разделу “Поиск каталога (папки) `learn`” этого приложения.

2. С помощью текстового редактора создайте файл исходного текста. С помощью `vi`, `ee` или какого-нибудь другого вашего любимого текстового редактора создайте и сохраните файл исходного текста. В качестве примера исходного текста вы можете взять распечатку `GOODBYE.C` из главы 1 “Основы языка C”; напечатайте этот текст с помощью вашего редактора.

3. Скомпилируйте исходный текст и отредактируйте связи.

Компиляцию и редактирование связей (оба действия) выполняет команда `GCC`. Вот что, например, вы должны напечатать, чтобы скомпилировать исходный текст `GOODBYE.C`, созданный в шаге 1, и затем скомпоновать его:

```
gcc goodbye.c -o goodbye
```

Эта команда состоит из четырех элементов:

- команды `gcc`, которая компилирует исходный текст и редактирует связи;
- названия (имени) файла исходного текста `goodbye.c`;
- переключателя вывода `-o`;
- названия (имени) полученной программы `goodbye`.

Если вы опустите переключатель `-o` и его опцию, в Windows `GCC` создаст программный файл по имени `a.out`. Я не рекомендую это. Воспользуйтесь опцией `-o` и укажите название (имя) выходной программы. Название (имя) может быть то же самое, что и у файла исходного текста, но без расширения `.c`.

4. Выполните полученную программу.

Увы, ваша операционная система не выполняет программу, если вы напечатали ее название (имя) после подсказки. Дело в том, что Unix выполняет только программы, найденные по указанному пути, а я не рекомендую помещать в путь ваш каталог `learn`. (Чтобы выполнить созданную программу, скопируйте ее в каталог `bin`, являющийся подкаталогом вашего основного каталога, и поместите *этот* каталог в путь.)

Чтобы операционная система смогла найти вашу программу, вы должны указать ее местонахождение (в текущей папке, например). Чтобы сделать это, перед названием (именем) программы нужно приписать `./`. Чтобы запустить программу `goodbye`, напечатайте следующее после подсказки:

```
./goodbye
```

и программа после этого будет выполнена.

Это основные шаги, которые нужно выполнить (все в папке `learn`), чтобы создать программы, приведенные в книге. В конечном счете вы будете выполнять их автоматически.



- ✓ Расширения имен файлов являются необязательными в Unix, но на самом деле я рекомендую использовать расширение `.c` для всех файлов исходных текстов на языке C. Это облегчит их хранение. В качестве награды, редактор `vim`, `GCC` и другие программы распознают `.c` и обрабатывают файл должным образом.
- ✓ Для каждой создаваемой программы есть два файла: файл исходного текста (текстовый файл) и программный файл.
- ✓ Компилятор `GCC` автоматически устанавливает биты разрешений для окончательного программного файла, позволяя его выполнять. Обычно это `--rwxr-xr-x`, или эквивалент команды `chmod 755`.
- ✓ Я рекомендую сохранять программы для возможности получения справки в будущем; поэтому не удаляйте их, пока не освоите программирование хорошо и действительно больше не будете нуждаться в них.

Приложение Б

Таблица ASCII

Код	Символ	Шестнадцатеричный код	Двоичный код	Примечание
0	^@	00	0000 0000	Нулевой символ (Null character), \0
1	^A	01	0000 0001	
2	^B	02	0000 0010	
3	^C	03	0000 0011	
4	^D	04	0000 0100	Клавиша выхода — Exit key (Unix)
5	^E	05	0000 0101	
6	^F	06	0000 0110	
7	^G	07	0000 0111	Звонок (Bell), \a
8	^H	08	0000 1000	Возврат на один символ (Backspace), \b
9	^I	09	0000 1001	Позиция табуляции (Tab), \t
10	^J	0A	0000 1010	Вертикальная позиция табуляции (Vertical tab), \v
11	^K	0B	0000 1011	
12	^L	0C	0000 1100	Перевод страницы (Form feed), \f
13	^M	0D	0000 1101	Клавиша <Enter>, \n (или \r)
14	^N	0E	0000 1110	
15	^O	0F	0000 1111	
16	^P	10	0001 0000	
17	^Q	11	0001 0001	
18	^R	12	0001 0010	
19	^S	13	0001 0011	
20	^T	14	0001 0100	
21	^U	15	0001 0101	
22	^V	16	0001 0110	
23	^W	17	0001 0111	
24	^X	18	0001 1000	
25	^Y	19	0001 1001	
26	^Z	1A	0001 1010	Конец файла (End of file) (в DOS)
27	^[1B	0001 1011	Escape
28	^\	1C	0001 1100	
29]`	1D	0001 1101	
30	^^	1E	0001 1110	
31	^_	1F	0001 1111	
32		20	0010 0000	Пробел (Space)

Код	Символ	Шестнадцатеричный код	Двоичный код	Примечание
33	!	21	0010 0001	
34	"	22	0010 0010	
35	#	23	0010 0011	
36	\$	24	0010 0100	
37	%	25	0010 0101	
38	&	26	0010 0110	
39	'	27	0010 0111	
40	(28	0010 1000	
41)	29	0010 1001	
42	*	2A	0010 1010	
43	+	2B	0010 1011	
44	,	2C	0010 1100	
45	-	2D	0010 1101	
46	.	2E	0010 1110	
47	/	2F	0010 1111	
48	0	30	0011 0000	(Цифры)
49	1	31	0011 0001	
50	2	32	0011 0010	
51	3	33	0011 0011	
52	4	34	0011 0100	
53	5	35	0011 0101	
54	6	36	0011 0110	
55	7	37	0011 0111	
56	8	38	0011 1000	
57	9	39	0011 1001	
58	:	3A	0011 1010	
59	;	3B	0011 1011	
60	<	3C	0011 1100	
61	=	3D	0011 1101	
62	>	3E	0011 1110	
63	?	3F	0011 1111	
64	@	40	0100 0000	
65	A	41	0100 0001	(Буквы верхнего регистра)
66	B	42	0100 0010	
67	C	43	0100 0011	
68	D	44	0100 0100	
69	E	45	0100 0101	
70	F	46	0100 0110	
71	G	47	0100 0111	

Код	Символ	Шестнадцатеричный код	Двоичный код	Примечание
72	H	48	0100 1000	
73	I	49	0100 1001	
74	J	4A	0100 1010	
75	K	4B	0100 1011	
76	L	4C	0100 1100	
77	M	4D	0100 1101	
78	N	4E	0100 1110	
79	O	4F	0100 1111	
80	P	50	0101 0000	
81	Q	51	0101 0001	
82	R	52	0101 0010	
83	S	53	0101 0011	
84	T	54	0101 0100	
85	U	55	0101 0101	
86	V	56	0101 0110	
87	W	57	0101 0111	
88	X	58	0101 1000	
89	Y	59	0101 1001	
90	Z	5A	0101 1010	
91	[5B	0101 1011	
92	\	5C	0101 1100	
93]	5D	0101 1101	
94	^	5E	0101 1110	
95	_	5F	0101 1111	
96	`	60	0110 0000	
97	a	61	0110 0001	(Буквы нижнего регистра)
98	b	62	0110 0010	
99	c	63	0110 0011	
100	d	64	0110 0100	
101	e	65	0110 0101	
102	f	66	0110 0110	
103	g	67	0110 0111	
104	h	68	0110 1000	
105	i	69	0110 1001	
106	j	6A	0110 1010	
107	k	6B	0110 1011	
108	l	6C	0110 1100	
109	m	6D	0110 1101	
110	n	6E	0110 1110	

Код	Символ	Шестнадцатеричный код	Двоичный код	Примечание
111	o	6F	0110 1111	
112	p	70	0111 0000	
113	q	71	0111 0001	
114	r	72	0111 0010	
115	s	73	0111 0011	
116	t	74	0111 0100	
117	u	75	0111 0101	
118	v	76	0111 0110	
119	w	77	0111 0111	
120	x	78	0111 1000	
121	y	79	0111 1001	
122	z	7A	0111 1010	
123	{	7B	0111 1011	
124		7C	0111 1100	
125	}	7D	0111 1101	
126	~	7E	0111 1110	
127		7F	0111 1111	Удалить (Delete) или "стереть"

Предметный указатель

"

", символ, 278

#

#define, директива, 104; 270; 272; 274

#else, директива, 276

#endif, директива, 276

#if, директива, 276

#include, директива, 269; 270; 273

#line, директива, 276

%

%%, символ преобразования, 282

%c, символ преобразования, 281

%d, символ преобразования, 281

%e, символ преобразования, 281; 282

%f, символ преобразования, 281

%g, символ преобразования, 281; 282

%i, символ преобразования, 281

%o, символ преобразования, 281

%s, символ преобразования, 282

%u, символ преобразования, 282

%x, символ преобразования, 282

&

&&, оператор, 169; 171; 172

&, оператор, 171

-

--, оператор, 192; 193

|

|, оператор, 171

||, оператор, 168; 169; 171

+

++, оператор, 188; 192; 193

A

abs(), функция, 288

acos(), функция, 288

AND, оператор, 169; 170; 171; 172

arccos(), функция, 288

arcsin(), функция, 288

arctg(), функция, 288

ASCII.C, программа, 181; 182

asin(), функция, 288

atan(), функция, 288

atoi(), функция, 84

B

B, язык программирования, 21

Backspace, символ, 278; 333

BASIC, язык программирования, 22

break, ключевое слово, 185; 186; 214; 215;
216; 221; 222; 224

C

C++, язык программирования, 21

case, ключевое слово, 220; 221; 222; 224; 227

cat, команда, 314

cc, 324

cd, команда, 314; 325; 327

char, числовой тип, 108; 110

clear, команда, 314

cls, команда, 314

CMD.EXE, 326

Code Warrior, 324

continue, ключевое слово, 214; 215; 216

cos(), функция, 288

COUNTDOWN.C, программа, 206; 208; 209;
210; 212

D

DBLQUOTE.C, программа, 52

default, ключевое слово, 220; 221; 222; 224

del, команда, 315

Delete, 336
Dev-C++, 328; 329
dir, команда, 314
do, ключевое слово, 207
double, числовой тип, 108
do-while, цикл, 206; 207; 208

E

Easy Editor, 327
ee, 327
else, ключевое слово, 150; 151; 152; 156; 222
End of file, 333
Escape, 333
Escape-последовательность, 278
Exit key, 333
exit, команда, 315
exp(), функция, 288

F

fflush(), функция, 161
float, числовой тип, 108
for, цикл, 175; 176; 177; 178; 179; 198; 200; 207
Form feed, символ, 278; 333
fputc(), функция, 162

G

GCC, 286; 323; 324; 328
GCC.BAT, 329
getch(), функция, 162
getchar(), функция, 123; 124; 159; 160; 161; 162
getche(), функция, 162
gets(), функция, 71; 72; 74; 122
GNU, 324
GOODBYE.C, программа, 26; 28
goto, ключевое слово, 174
grep, программа, 321

I

if, ключевое слово, 141; 142; 144; 156; 157; 164; 166
INCLUDE, каталог, 269; 270; 274
int, числовой тип, 108; 109; 110

J

JUSTIFY.C, программа, 55

L

lg(), функция, 288
LIBRARY, каталог, 286
lint, программа, 321
log(), функция, 288
log10(), функция, 288
long int, числовой тип, 109; 110
long, числовой тип, 108; 109; 110
ls, команда, 314

M

main(), функция, 262; 263
make, программа, 321
MDAS, 137
Microsoft Visual C++, 324
mkdir, 325
MSVC, 324
mv, команда, 314
My Dear Aunt Sally, 137; 138
My Documents, 324

N

newline, символ, 278
Notepad, 328
NULL, 306
Null character, 333

O

OR, оператор, 168; 169; 170; 171

P

pow(), функция, 284; 285; 286
printf(), функция, 51; 54; 76; 277; 278; 280
PRINTFUN.C, программа, 279; 280
putchar(), функция, 123; 124
puts(), функция, 73; 74; 75; 76
pwd, команда, 314; 327

R

rand(), функция, 294; 296
RAND_MAX, 295

RANDOM1.C, программа, 294
RANDOM2.C, программа, 296
RANDOM3.C, программа, 298
RANDOM4.C, программа, 301; 302
rep, команда, 314
return, ключевое слово, 259; 261; 262
rm, команда, 315

S

scanf(), функция, 57; 122
seedrnd(), функция, 297; 302
short int, числовой тип, 108; 109
short, числовой тип, 108; 109
sin(), функция, 288
sleep(), функция, 212; 213
Space, 333
sqrt(), функция, 286; 287
srand(), функция, 296
STDIO.H, файл, 270
strcmp(), функция, 306
struct, ключевое слово, 307
switch, ключевое слово, 220; 221; 222; 224
switch-case, переключатель, 217; 219; 220;
221; 222; 224; 227

T

Tab, 333
tan(), функция, 288
tg(), функция, 288
time(), функция, 299
type, команда, 314
TYPED1.C, программа, 184; 185

U

unsigned char, числовой тип, 108; 110
unsigned int, числовой тип, 108; 110; 111
unsigned long, числовой тип, 108; 110; 111
unsigned short, числовой тип, 108

V

Vertical tab, 333
vi, 327
Vim, 312; 327

W

while, ключевое слово, 207
while, цикл, 199; 207; 224

A

Абсолютное значение, 288
Алгол-68, язык программирования, 25
Алмир, язык программирования, 25
Аналитик, язык программирования, 25
Апостроф, символ, 278
Арифметическая прогрессия, 194
Аркосинус, функция, 288
Арсинус, функция, 288
Артангенс, функция, 288

Б

БЕЙСИК, язык программирования, 22
Бесконечный цикл, 182; 199; 202
Библиотека, 274
Библиотечный файл, 273
Биты разрешений, 332
Блокнот, 328

В

Ввод символов, 120
Вертикальная позиция табуляции,
символ, 278; 333
Возведение в степень, 285
Возврат на один символ, 278; 333
Вопросительный знак, символ, 278
Выполнение, 331
Выход из цикла, 315

Г

Генератор случайных чисел, 295; 298
Глобальная переменная, 247
Гудок динамика, символ, 278

Д

Двойная кавычка, символ, 278
Декремент, 289
Десятичный логарифм, 288
Диапазон чисел с плавающей точкой, 116
Директива

#define, 104; 270; 272; 274
#else, 276
#endif, 276
#if, 276
#include, 269; 270; 273
#line, 276

Документация, 320

Доступ к файлам на диске, 310

З

Заголовочный файл, 269; 272

Заполнитель, 56

Звонок, 333

Знаки математических операций, 130

Знаки операций сравнения, 145

И

И, оператор, 169; 170; 171; 172

Извлечение квадратного корня, 286

Извлечение корня, 286

ИЛИ, оператор, 168; 169; 170; 171

Имена переменных, 96

Инкремент, 133; 289

Исполнимая программа, 274

Исправление ошибок, 317

Исходный текст, 24

К

Каталог

INCLUDE, 269; 270; 274

LIBRARY, 286

Квадратный корень, 286

Ключевое слово, 42; 43

break, 185; 186; 214; 215; 216; 221;
222; 224

case, 220; 221; 222; 224; 227

continue, 214; 215; 216

default, 220; 221; 222; 224

do, 207

else, 150; 151; 152; 156; 222

goto, 174

if, 141; 142; 144; 156; 157; 164; 166

return, 259; 261; 262

struct, 307

switch, 220; 221; 222; 224

while, 207

Команда

cat, 314

cd, 314

clear, 314

cls, 314

del, 315

dir, 314

exit, 315

ls, 314

mv, 314

pwd, 314

ren, 314

rm, 315

type, 314

Комментарий, 63; 320

в C++, 67

вложенный, 68

как средство отключения кода, 68

стили, 66

суперкомментарии, 66

Компилятор, 270; 323

Конец файла, 333

Константы, 101

Контрольные точки, 319

Косинус, функция, 288

Л

Логарифм по основанию 10, функция, 288

Логические выражения, 168

Логические операции, 171

Локальная переменная, 247

М

Макрокоманда, 275

Макроопределение, 270; 275

Макрос, 270; 275

Массив, 305; 306

Математическая библиотека, 286

Математические символы, 120

Меню, 157; 163

Метка-заполнитель, 56

Множественные объявления, 101

Модуль, 299

Мои документы, 324

Моя Дорогая Тетушка Салли, 137; 138

Н

Наклонная черта влево, символ, 278
Натуральный логарифм, 288
Неположительные числа, 81
Новая строка, символ, 278
Нулевой символ, 333

О

Обсуждение программы, 319
Объектный код, 273
Объектный файл, 274
Объявление переменных, 96
Односимвольные переменные, 119

Оператор

&, 171

&&, 169; 171; 172

!, 171

!!, 168; 169; 171

++, 188; 192; 193

+++, 193

AND, 169; 170; 171; 172; 309

EOR, 309

Exclusive OR, 309

Inclusive OR, 309

OR, 168; 169; 170; 171; 309

XOR, 309

включительное ИЛИ, 309

двоичный, 308

декремента, 192

дополнение, 309

И, 169; 170; 171; 172; 309

ИЛИ, 168; 169; 170; 171; 309

исключительное ИЛИ, 309

неэквивалентность, 309

побитовый сдвиг влево, 309

побитовый сдвиг вправо, 309

поразрядное дополнение, 309

приращения, 188

Операционная система, 310

Определение констант, 102

Оптимизатор, 321

Остаток от деления, 299

Отладчик, 321

Отсчет в обратном порядке, 190; 191

Ошибки, 34; 38

дефекты, 38

исправление, 35
критические, 36
неустраняемые, 36
обнаруживаемые во время выполнения программы, 38
обнаруживаемые компилятором, 38
обнаруживаемые компоновщиком, 38
предупреждения, 36
причины, 34
фатальные, 36

П

Параметр командной строки, 309

Перевод каретки, символ, 278

Перевод страницы, 278; 333

Переключатель

switch, 223

switch-case, 217; 219; 220; 221; 222;
224; 227

Переменная, 79; 101; 246; 320

глобальная, 247; 248; 250

именование, 315

локальная, 247

создание глобальной переменной, 247

Позиция табуляции, 278; 333

Показательная функция, 288

Постинкремент, 291

Постоянные, 105

Постприращение, 291

Предварительная установка значений
переменных, 97

Предприращение, 291

Предупреждения, 36

Пресинкремент, 291

Препроцессор, 270

Приведение типа, 261

Приращение, 133; 188

Пробел, 333

Программа

ASCII.C, 181; 182

COUNTDOWN.C, 206; 208; 209; 210; 212

DBLQUOTE.C, 52

GOODBYE.C, 26; 28; 39

grep, 321

JUSTIFY.C, 55

lint, 321
make, 321
PRINTFUN.C, 279; 280
RANDOM1.C, 294
RANDOM2.C, 296
RANDOM3.C, 298
RANDOM4.C, 301; 302
RULES, 45
TYPER1.C, 184; 185
WHORU.C, 49
защищенная, 209
ПРАВИЛА, 45
пуленепробиваемая, 209
составные части, 40
текстовый процессор, 184
Псевдослучайное число, 294

Р

Разбиение на модули, 318
Разработка больших программ, 310
Редактирование связей, 328; 330; 331
Редактор, 327

С

Связный список, 308
Символ
", 278
\\, 278
_хе, 278
Backspace, 278
Form feed, 278
newline, 278
апостроф, 278
вертикальная позиция табуляции, 278
возврат на один символ, 278
вопросительный знак, 278
гудок динамика, 278
двойная кавычка, 278
наклонная черта влево, 278
новая строка, 278
перевод каретки, 278
перевод страницы, 278
позиция табуляции, 278
преобразования, 56
%%, 282
%с, 281

%d, 281
%е, 281; 282
%f, 281
%g, 281; 282
%i, 281
%о, 281
%s, 282
%u, 282
%x, 282

табуляции, 121

Символы псевдографики, 120
Синус, функция, 288
Случайность, 293
Стандартная библиотека, 286
Стандартная математическая библиотека, 286
Старшинство операций, 136
Степень, 284
Строка, 306
Строка формата, 281
Структура, 307; 308

Т

Тангенс, функция, 288
Текстовый редактор, 184

У

Указатель, 308
Упрощение, 318
Условный оператор, 147

Ф

Файл
STDIO.H, 270
исходного текста, 24
Фатальные ошибки, 36
Форматирование, 282
десятичного числа, 117
Функциональный стиль, 202
Функция, 231; 251
abs(), 288
acos(), 288
arccos(), 288
arcsin(), 288
arctg(), 288
asin(), 288
atan(), 288

atoi(), 84
cos(), 288
exp(), 288
fflush(), 161
fpurge(), 162
getch(), 162
getchar(), 123; 124; 159; 160; 161; 162
getche(), 162
gets(), 71; 72; 74; 122
lg(), 288
log(), 288
log10(), 288
main(), 262; 263
pow(), 284; 285; 286
printf(), 51; 54; 76; 277; 278; 280
putchar(), 123; 124
puts(), 73; 74; 75; 76
rand(), 294; 296
scanf(), 57; 122
seedrnd(), 297; 302
sin(), 288
sleep(), 212; 213
sqrt(), 286; 287
srand(), 296
strcmp(), 306
tan(), 288
tg(), 288
time(), 299
абсолютное значение, 288
аргумент, 253
арккосинус, 288
арксинус, 288
арктангенс, 288
возврат значения, 253; 261
возвращаемое значение, 262
главная, 241
десятичный логарифм, 288
именование, 241
имя, 239; 241
косинус, 288
логарифм по основанию 10, 288
название, 241
натуральный логарифм, 288
объявление, 240
определение, 240

параметры, 240; 253; 255
передача значения, 251; 252; 253
передача нескольких значений, 256
показательная, 288
прототип, 237; 238; 240; 252
синус, 288
создание прототипа, 236
тангенс, 288
тип, 239
формат, 239

Ц

Целое число, 81
Целочисленные типы, 109
Цикл, 173; 174
do-while, 206; 207; 208
for, 175; 176; 177; 178; 179; 198;
200; 207
while, 197; 199; 207; 224
бесконечный, 182; 183
внутри цикла, 210
выбор типа, 199; 200
задержки, 212
замена типа цикла, 201
инструкция, 177
начало, 176; 177
пока, 197
приращение, 176; 177
прыгающий, 194
с условием продолжения, 197; 198; 200;
205; 206; 207; 227
управление, 200
условие, 176; 178; 199; 200; 207
условие выхода, 174; 184
Цикл разработки, 22

Ч

Числа, 108
Числовой тип, 108
char, 108; 110
double, 108
float, 108
int, 108; 109; 110
long, 108; 109; 110

long int, 109; 110
short, 108; 109
short int, 108; 109
unsigned char, 108; 110
unsigned int, 108; 110; 111
unsigned long, 108; 110; 111
unsigned short, 108

Ш

Ширина, 56

Э

Экспоненциальное представление, 114

Я

Язык программирования

В, 21
BASIC, 22
C++, 21
Алгол-68, 25
Алмир, 25
Аналитик, 25
БЕЙСИК, 22



TM

BESTSELLING
BOOK
SERIES

С для "чайников"™ 2-е издание



TM

СЕРИЯ
КОМПЬЮТЕРНЫХ
КНИГ ОТ
ДИАЛЕКТИКИ

Знаки операций сравнения

Значение или чтение	Примеры верных неравенств
< меньше	1<5 8<9
== равно	5==5 0==0
> больше	8>5 10>0
<= меньше или равно	4<=5 8<=8
>= больше или равно	9>=5 2>=2
!= не равно	1!=0 4!=3.99

Escape-последовательности

Последовательность	Представляет
\a	подача звукового сигнала динамика
\b	возврат (Backspace) на один символ (перемещает курсор назад без стирания)
\f	перевод страницы (Form feed) (выдает страницу принтера; специальный символ на экране)
\n	новая строка (newline), подобно нажатию клавиши <ENTER>
\r	перевод каретки (carriage return) (перемещает курсор в начало строки)
\t	табуляция (tab)
\v	вертикальная табуляция (vertical tab) (перемещает курсор вниз на одну строку)
\l	символ наклонной черты влево
\'	апостроф
\"	символ двойной кавычки
\?	вопросительный знак
\0	"нулевой" байт (ноль после наклонной черты влево)
\xnnn	шестнадцатеричное представление символического значения (по основанию 16)
\Xnnn	шестнадцатеричное представление символического значения (по основанию 16)

Шпаргалка

Символы преобразования

Символ преобразования	Отображает параметр (значение переменной) как зования
%c	один символ
%d	десятичное целое число со знаком (int)
%e	значение с плавающей точкой со знаком в экспоненциальном представлении
%f	значение с плавающей точкой со знаком (float)
%g	значение со знаком в формате %e или %f, в зависимости от того, какой из них короче
%i	десятичное целое число со знаком (int)
%o	восьмеричное целое число без знака (по основанию 8) (int)
%s	строка текста
%u	десятичное целое число без знака (int)
%x	шестнадцатеричное целое число без знака (основание 16) (int)
%%	(символ процента)

Операции сравнения и их отрицания

Операция сравнения в if	Инструкция else выполняется при этом условии
<	>= (больше или равно)
==	!= (не равно)
>	<= (меньше или равно)
<=	> (больше)
>=	< (меньше)
!=	-- (равно)



BESTSELLING
BOOK
SERIES

С для "чайников"™ 2-е издание



СЕРИЯ
КОМПЬЮТЕРНЫХ
КНИГ ОТ
ДИАЛЕКТИКИ

Ключевые слова языка C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	static	void
default	goto	sizeof	volatile
do	if	signed	while

Шпаргалка

Математические знаки

Оператор	Что означает	Читается как	Выполняемая операция
+	+	"плюс"	сложение
-	-	"минус"	вычитание
*	*	"умножить"	умножение
/	:	"разделить"	деление
%	mod	"остаток по модулю"	остаток по модулю

Типы числовых данных

Ключевое слово	Тип переменной	Диапазон
char	символ (или строка)	от -128 до 127
int	целое число	от -32 768 до 32 767
short	короткое целое число	от -32 768 до 32 767
short int		
long	длинное целое число	от -2 147 483 648 до 2 147 483 647
unsigned char	символ без знака	от 0 до 255
unsigned int	целое число без знака	от 0 до 65 535
unsigned short	короткое целое число без знака	от 0 до 65 535
unsigned long	длинное целое число без знака	от 0 до 4 294 967 295
float	с плавающей точкой (с плавающей запятой) с одинарной точностью (точность 7 цифр)	от $\pm 3,4 \times 10^{-38}$ до $\pm 3,4 \times 10^{38}$
double	с плавающей запятой с двойной точностью (точность 15 цифр)	от $\pm 1,7 \times 10^{-308}$ до $\pm 1,7 \times 10^{308}$

Научно-популярное издание

Дэн Гукин

С для "чайников"

2-е издание

В издании использованы карикатуры американского художника Рича Теннанта

Литературный редактор	<i>П.Н. Мачуга</i>
Верстка	<i>О.В. Мишутина</i>
Художественный редактор	<i>В.Г. Павлютин</i>
Корректоры	<i>Л.А. Гордиенко, О.В. Мишутина</i>

Издательский дом "Вильямс".
101509, г. Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 05.06.2006. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 20,0. Уч.-изд. л. 19,2.
Доп. тираж 3000 экз. Заказ № 1639.

Отпечатано с диапозитивов
в ОАО "Печатный двор" им А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., 15.

C++ для "чайников"

5-е издание

С. Дэвис



ISBN 5-8459-0723-3

В продаже

Книга представляет собой введение в язык программирования C++. Основное отличие данной книги от предыдущих изданий C++ для "чайников" в том, что это издание не требует от читателя каких-либо дополнительных знаний, в то время как предыдущие издания опирались на знание читателем языка программирования C. Книга отличается также тем, что несмотря на простоту изложения материала, он подан достаточно строго. Поэтому, изучив основы программирования на C++, читателю не придется пересматривать свои знания при дальнейшем изучении языка.

Книга отличается широким охватом тем — от простейших объявлений переменных до концепций объектно-ориентированного программирования, вопросов перегрузки операторов и множественного наследования, причем весь материал снабжен множеством практических примеров.

Эта книга не учит программированию в Windows или созданию красивого интерфейса двумя движениями мышь; изложенный в ней материал не привязан к какому-то определенному компилятору или операционной системе. Она вряд ли будет полезна профессиональному программисту, но если ваша цель — глубокое знание языка программирования и вы не знаете, с чего начать — эта книга для вас.

ОСНОВЫ ПРОГРАММИРОВАНИЯ ДЛЯ "ЧАЙНИКОВ" 3-Е ИЗДАНИЕ

Уоллес Вонг



www.dialektika.com

Простым человеческим языком автор книги расскажет вам, что такое программирование вообще, что такое язык программирования, как создаются компьютерные программы и какие инструменты для этого используются. Большое внимание в книге уделяется одному из самых популярных и простых для изучения языков программирования — BASIC. Кроме того, рассмотрены и такие языки программирования, как C/C++, C#, Pascal, Delphi, JBuilder, Java, JavaScript и некоторые другие. Вы узнаете, как организовать ввод и вывод данных, как использовать в программах переменные, константы и комментарии, познакомитесь с основными правилами использования чисел и строк, управляющими конструкциями и циклами. Книга рассчитана на пользователей с начальным уровнем подготовки.

ISBN 5-8459-0690-3

в продаже

ЦИФРОВОЕ ВИДЕО ДЛЯ "ЧАЙНИКОВ", 3-е издание

Кит Андердал



www.dialektika.com

Цифровое видео — это революция, которая изменяет само представление о способах создания и использования видеофильмов. Настоящая книга не только поможет вам в освоении этой новой технологии или каких-то конкретных программ видеомонтажа. В ней рассказывается об искусстве создания видеофильмов, о том, как можно использовать эту новую великолепную технологию для создания собственного потрясающего видео. Эта книга была задумана как общее руководство по созданию видеофильмов. Из ее разделов вы узнаете, как подобрать хорошую видеокамеру, как отснять качественный видеоматериал, разместить фильмы в сети Internet, а в беседах на подобные темы проявлять такой уровень эрудиции, будто вы работаете в Голливуде.

в продаже

МОДЕРНИЗАЦИЯ И РЕМОНТ ПК ДЛЯ "ЧАЙНИКОВ", 6-е издание

Энди Ратбон



Хотите, чтобы ваш компьютер делал больше, а ломался реже, но не знаете, с чего начать? Расслабьтесь! Это полностью обновленное издание известной книги снимет покров тайны с современного программного и аппаратного обеспечения — начиная с видеокарт и Windows XP и заканчивая стандартом FireWire и брандмауэрами, — и шаг за шагом расскажет о том, как превратить обычный серенький компьютер в безотказную супермощную мультимедиа-машину. Книга предназначена для начинающих пользователей.

www.dialektika.com

в продаже

УСТРАНЕНИЕ НЕИСПРАВНОСТЕЙ В ПК ДЛЯ "ЧАЙНИКОВ" 2-Е ИЗДАНИЕ

Дэн Гукин

Эта книга является понятным руководством по получению информации об устранении и предупреждении неисправностей в ПК. Тем, кто заблудился в дебрях Windows и не знает, как избавиться от надоевшего "глюка", можно дать один простой совет — откройте эту книгу, и все сразу станет на свои места. Кроме того, книга написана простым и доступным языком, без излишних технических подробностей, которые все равно поняли бы только специалисты. Вместо них автор приводит четкие инструкции, следуя которым, можно устранить любую неисправность, связанную с монитором, клавиатурой, мышью, принтером, модемом и т.п. Книга предназначена для широкого круга читателей.



www.dialektika.com

ISBN 5-8459-0999-6

в продаже

Научитесь

управлять

компьютером



Разработка программ,

компиляция

и редактирование связей,

устранение проблем

и многое другое...

Сами убедитесь, насколько удобен язык C! На C легко писать быстрые, компактные и к тому же универсальные программы, притом в мгновение ока! В этой дружественной читателю книге раскрываются секреты кодирования и комментирования программ, их компиляции, а также использования ключевых слов языка, системы ввода-вывода, переменных, целых чисел и всего остального. Вы напишете свою первую программу на C, даже не дочитав первую главу!

В стиле
для
"ЧАЙНИКОВ"

- Объяснения простым и доступным языком
- Информация "без лишних подробностей"
- Пиктограммы и другие средства ориентирования в материале книги
- Шпаргалка, включающая самую ценную информацию
- Великолепные десятки советов
- Много юмора и шуток

Категория:
языки программирования/C

Уровень:
для начинающих пользователей

Посетите "Диалектику" в Internet по адресу <http://www.dialektika.com>

**Эта книга
поможет вам:**

- Научиться создавать программы
- Редактировать связи для создания исполнимых программ
- Отлаживать программы
- Использовать целочисленные переменные и переменные с плавающей запятой, а также условные операторы
- Создавать функции и применять циклы

Дэн Гукин в 1991 году написал первую книгу серии *For Dummies — DOS For Dummies*. С тех пор его книги были переведены на 32 языка, и их общий тираж превысил 11 миллионов.

ДИАЛЕКТИКА
For Dummies®
A Branded Imprint of
WILEY